Deliverable 5.3: Final Report on Hardware/Software Integration and Robotics Test Bed

WP5 *

December 21, 2016

* The TWO!EARS project (http://www.twoears.eu) has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 618075.

Project acronym:	Two!Ears
Project full title:	Reading the world with Two!Ears

Work package:	WP5
Document number:	D5.3
Document title:	Final Report on Hardware/Software Integration and Robotics Test Bed
Version:	1

Delivery date:	12th December 2016
Actual publication date:	12th December 2016
Dissemination level:	Public
Nature:	Report

Editor:	Patrick Danès
Author(s):	Sylvain Argentieri, Gabriel Bustamante, Benjamin Cohen-L'Hyver, Patrick Danès, Xavier Dollat, Thomas Forgue, Bruno Gas, Matthieu Herrb, Anthony Mallet, Jérôme Manhès, Antonyo Musabini, Jonathan Piat, Ariel Podlubne, Bertrand Vandeportaele
Reviewer(s):	Armin Kohlrausch

Contents

1 Executive summary

The computational framework of auditory perception and experience designed in Two!Ears is implemented as a *development* software system primarily based on *MATLAB*. The evaluation of the Two!Ears model for different live scenarios implies a *deployment* system, consisting in the interface of the development system with a robot. Work package WP5 is in charge of all the necessary ingredients of this deployment. To assess the active and exploratory features of the computational model and its ability to handle multimodality, two robot platforms endowed with adequate mobility and multimodal perception were designed. Each of them is accompanied by a comprehensive real time software architecture, entailing a modular low "functional" layer, where components run concurrently under severe time and communication constraints, and a high "cognitive" layer, where decisional processes take place. All the functional modules were systematically submitted to extensive tests.

This deliverable summarizes the achievements of work package WP5 towards the deployment of the two robotics test beds. Each one consists in the mounting on a differential wheeled robot of the *KEMAR* head-and-torso simulator endowed with a controllable neck degree-of-freedom. By adding a specifically designed anthropomorphic stereoscopic visual sensor, these platforms can provide translational degrees-of-freedom for long-range navigation as well as multimodality. The comprehensive, stable, companion real time software architecture was implemented along the following guidelines: the selection of the celebrated *ROS*[1] middleware; the use of off-the-shelf *ROS*-compliant software iff it is suitable and has been successfully tested by the robotics community; the design of components specific to the project by means of the model-driven middleware-independent *GenoM3*[2] framework, for an improved robustness, sustainability, and code reusability. Its prominent elements are described, namely: an improved, self-sufficient, *MATLAB* bridge to connect the functional layer with multiple *MATLAB* cognitive processes; custom integrations of the low-level functions for locomotion and sensor handling under standard interfaces, so as to enable a transparent interchange of the robot; off-the-shelf widely used software for simultaneous localization and mapping (SLAM) and planned/reactive navigation; components for audio and visual streaming; transcoding of a significant part of the *MATLAB* based auditory front-end (AFE) into a *ROS* component for real time concurrent processing; sensory/sensorimotor functions for multiple people detection/tracking,

1 *R*obot *O*perating *S*ystem, `http://www.ros.org` – This open-source meta-operating system has been initiated by Willow Garage, and runs on the top of Linux.

2 *Gen*erator *o*f *M*odules *v3*, `https://git.openrobots.org/projects/genom3/wiki/Wiki` – This framework is one of the core software component distributed within the open-source collection developed at CNRS, as a result of two decades of research on real-time architectures for autonomous systems.

learning/detection/segmentation of objects, and binaural active azimuth+range sound source localization.

The manuscript is intended to be synthetic—in that details available in Deliverables D5.1@m12 and D5.2@m24 are ommitted—but self-contained. To limit its size, the low-cost integrated binaural sensors (MEMS microphones, spherical binaural heads, *ROS*-compliant system-on-chip acquisition and streaming) specifically designed and succesfully deployed for the "robotics challenge" of the TWO!EARS Summer School (September 2015) are not described. The interested reader is referred to Deliverables D5.2@m24.

2 Introduction

The main objective of WP5 is to integrate the whole set of modules from WPs2–4 into a physical test bed which enables the global evaluation of the TWO!EARS computational framework in scenarios of WP6. This has implied: the design and manufacturing of a suitable electromechanical device enabling the servocontrol of the neck of a *KEMAR* Head-And-Torso Simulator (HATS); the design and implementation of an anthropomorphic stereovision system perfectly fitted to the face of the *KEMAR* head; the mounting of two samples of motorized *KEMAR* HATSs on two differential wheeled mobile robots so as to offer translation degrees-of-freedom and enable long-range motions. A comprehensive modular real time software architecture has been deployed with this hardware. Its lower functional layer is made with components which run concurrently under severe time constraints and communicate by requests or data in real time. It is bridged with the upper cognitive layer. Therein, decisional processes take place, which handle symbolic data and are subject to less critical constraints. Extensive evaluations of all atomic elements have been conducted so as to ensure their satisfactory behaviour when case studies are addressed through the whole, integrated, deployment system.

2.1 Structure of the report and major achievements

The present document is organized as follows.

Chapter 3 recalls fundamental elements of the **robotics software architecture**. General considerations are reviewed on how to bridge the gap between, on the one hand, the TWO!EARS conceptual model, and, on the other hand, the functional and cognitive layer of the real time software architecture supporting the deployed test beds.

Chapter 4 reports the work conducted on TWO!EARS **test beds**. Hardware solutions and their companion low-level software components are depicted, based on a *KEMAR* Head-And-Torso Simulator. These include: binaural audio acquisition and streaming; the nonintrusive motorization of the neck for head rotational motions; the nonintrusive addition of a stereoscopic sensor for incorporation of multimodality; the assembly of two samples of the HATS with two mobile platforms with similar kinematics for exploratory navigation.

Chapter 5 describes the **transcoding of a subset of the Auditory Front End** (AFE, developed in WP2 for *MATLAB* based low-level audio processing) **into the real-time software architecture**. A new *C++* implementation is detailed, focusing on performance through concurrent processing.

Chapter 6 reports works conducted on **sensorimotor functions** for active sound source localisation, and **visual functions** for multiple people detection and tracking, as well as visual based learning, detection and localisation of objects.

Appendix 7 concludes the manuscript.

Note that ingredients specifically developed for the "robotics challenge" of the Two!Ears Summer School in September 2015 are not reviewed here, as they were extensively described in Chapter 7 of Deliverable 5.2@m24.

2.2 Structure of the report vs Tasks Decomposition

WP5 is split into three tasks. However, for easier readability, the manuscript is not organized along these. Rather, it is organized along the main achievements, starting from hardware and going to software.

- **Task 5.1 Test bed: robot platforms and integrated audio/audiovisual sensors**. A key achievement has concerned the deployment of two sustainable visio-auditive platforms suited to the experiments envisaged in the project. An existing robust mobile robot, named *Jido*, was remanufactured in depth at CNRS during Year 2, and another mobile robot named *Odi*, was specified by UPMC, and received during Year 3 (Section 4.4). The KEMAR head-and-torso simulator (HATS) with its motorized azimuth degree of freedom (Section 4.2), implemented during Year 1, was slightly revisited in Year 3 and mounted on both platforms. A stereoscopic sensor was installed on the KEMAR head (Section 4.3) of CNRS. It consists in the assembly of 3D-printed glasses designed from the head CAD model and of micro-cameras with suitable lenses. Drivers were specifically developed for the KEMAR HATS, including homing, proprioception and servocontrol. They were encapsulated into a software component for the aforementioned *ROS* middleware. Low-level libraries for locomotion, teleoperation, proprioceptive and exteroceptive sensing were also specifically developed for *Jido* and integrated in a custom *ROS* stack. *Odi* was delivered with a *ROS* stack ensuring similar functions. Both packages have the same standard interface (Section 4.5) required by higher-level off-the-shelf *ROS* stacks, so that *Jido* and *Odi* can be interchanged with no software change.

- **Task 5.2 Software architecture of the** Two!Ears **framework**. The real time software architecture supporting the deployment system has reached a mature, stable state. It is well suited to experiments on the basis of an auditory or audio-visual mobile robot. Its functional layer is built on the top of *ROS* (Section 3.1). It includes off-the-shelf widely-used open-source *ROS* stacks (change of frames handling, path planning, obstacle avoidance, reactive navigation, SLAM, stereovision from the used pair of micro-cameras) as well as components specifically designed under *GenoM3/ROS* for the needs of the project: audio acquisition (Section 4.1); control of the KEMAR head (Section 4.2); multiple people detection and tracking (Section 6.2); objects detection and segmentation (Section 6.3); audio-motor binaural localization

and sensorimotor feedback control (Section 6.1); transcoding of a significant part of the Auditory Front End (Section 5). The *MATLAB* bridge (Section 3.2), enabling the communication between the functional layer and one or multiple *MATLAB* processes implementing the cognitive layer, was fully revisited in Year 2 so as to be suited to *GenoM3/ROS* as well as native *ROS* components and to improve its overall performance.

- **Task 5.3 Modular tests and evaluations**. Extensive atomic tests have been conducted on the hardware and software described above. The outcomes of Tasks 5.1 and 5.2 reinforce possibilities of interfacing them with developments in work packages WP2, WP3, and WP4, so as to address challenging scenarios.

2.3 Overview of (visio-)auditive robotics test beds

This section briefly introduces the platforms supporting the Two!Ears deployment system (Figure 2.1).

Figure 2.1: From left to right: the *KEMAR* HATS; the initially planned *PR2* robot; the *Jido* and *Odi* Two!Ears test beds, respectively from *CNRS* and *UPMC*.

2.3.1 The *KEMAR* head-and-torso-simulator

The *KEMAR* (Knowles Electronics Manikin for Acoustic Research) Head-And-Torso Simulator (HATS) is widely used for reproducible human-like binaural acquisition (*e.g.*, to assess electroacoustic devices), thanks to its ability to mimic acoustic scattering and reflections on human upper bodies. It is based on worldwide average human male and female head and torso dimensions and meets the requirements of ANSI S3.36/ASA58-1985 and IEC 60959:1990. The *45BB-2* model used in the project is made up with a ruggedized plastic composite. Its two ears, which can be selected from six different types, can be accurately positioned and easily dismantled for ear-canal exchange or calibration.

2.3.2 The mobile platforms *Jido* (CNRS) and *Odi* (UPMC)

The mobile manipulator *PR2* from Willow Garage had initially been selected as the Two!Ears robotics test bed because of its openness, versatility (being undoubtedly the emblematic robot based on the *ROS* middleware), and dissemination (*e.g.*, *CNRS* and *UPMC* respectively own 2+1 units). However, during Year 1, a great number of failures were reported on the two *PR2* robots from *CNRS* (caster and arm control boards, EtherCAT hub, actuators, sensors, batteries,...). Furthermore, it appeared that the sporadic and unpredictable fan cooling of their four-caster bases induced a very loud noise. So, the *PR2* was discarded. It was replaced by two robots, namely a refurbished existing platform at *CNRS* named *Jido*, and a new platform at *UPMC* named *Odi*. Importantly, both test beds have similar kinematics, in that they are differential wheeled non-holonomic. Contrarily to what would have been possible with the *PR2*, they can carry the whole *KEMAR* HATS. As shown later, they can be accessed through the same standard interface, which enables reproducible research inside and outside the consortium.

3 The TWO!EARS deployment software architecture

The TWO!EARS computational model of auditory perception and experience entails low-level audio processing (developed in WP2), high-level feature extraction and reasoning (WP3), as well as various sorts of feedback (WP4). The *development system*, implemented in *MATLAB*, enables tests of these elements on simulated data, *e.g.*, generated in WP1. Work package WP5 is in charge of the synthesis of a *deployment system* enabling the confrontation of concepts and algorithms against real-life scenarios defined in WP6 and in connection with WP1. This deployment system is based on robotics test beds endowed with mobility and multimodality. It is grounded in a comprehensive generic software architecture, built on the top of their instrumentation and of the encapsulation of their basic capabilities into standard interfaces.

In this chapter, Section 3.1 first describes how the TWO!EARS conceptual framework is turned into a component-based deployment software. The celebrated *ROS* middleware and *GenoM3* framework constitute the cornerstone of the real time architecture supporting its lowest layer. The development system, designed in WP2,WP3,WP4 and written in *MATLAB*, is also hosted by the deployment software. A solution bridging both layers is exposed in Section 3.2. Last, installation and licensing aspects are evoked in Section 3.3.

3.1 From the conceptual framework to a component-based software architecture

3.1.1 Functional and decisional layers

From a robotics viewpoint, any comprehensive software architecture entails at least two layers:

- The *functional layer* consists of components which may run concurrently under severe time and communication constraints. These are in charge of sensory/sensorimotor functions, such as: locomotion; acquisition, streaming and low-level processing of proprioceptive or exteroceptive data; localisation; path planning; reactive navigation with obstacle avoidance; etc. As many components are in interaction with the environment, several local perception-action loops take place in this layer. Typical programming languages are *C* or *C++*. Components are implemented on the top of

a dedicated software called middleware, which ensures their real time control and communication.

- Higher in the architecture, the *decisional/cognitive layer* hosts deliberation primitives (learning, goal reasoning, task planning, deliberate action/perception and monitoring). These abilities take place at a more abstract level, under lighter time constraints. They are typically implemented under an interpreted language: symbolic reasoning system, supervisor, etc.

In TWO!EARS, the functional layer must basically provide components for audio (binaural) and visual (stereoscopic) data streaming, as well as for motion and navigation. The celebrated *ROS* middleware, exposed in Section 3.1.3, has been selected to support their control and communication. The cognitive part of WP3, as well as top-down hypothesis-driven feedbacks of WP4, are part of the development system and straightly take place within the decisional layer. So, quite uncommonly in a robotics context, this layer is written in *MATLAB*. An intermediate set of abilities can be identified in-between, which implementation can either come in *MATLAB* or in *C/C++* components, depending on the need for responsiveness. For instance, the Auditory Front-End (AFE) was developed in WP2 for monaural and binaural processing, and primarily written in *MATLAB*. In anticipation of large-scale experiments running on a single computer, part of it has been implemented in *C++* as a *ROS* component, see Section 5. Another example is related to visual functions. Low-level routines such as calibration or image rectification can be incorporated in the component in charge of streaming. More elaborated functions such as object detection and segmentation, or human detection and tracking, could be implemented in *MATLAB*. However, they must run at high frequencies and thus also come into dedicated components of the functional layer, see Sections 6.2 and 6.3.

To conclude, Figure 3.1 summarizes the way how the TWO!EARS model can be matched with a component-based robotics deployment software. A specific software bridge must be inserted between the upper decisional and lower functional layers, *i.e.*, between the *MATLAB*-based and *ROS*-based levels. It is presented in Section 3.2.

3.1.2 Component-based architecture, data and control flows

In robotics, *component-based architectures*, where components are concurrent and independent processes, have become the *de facto* standard. Each software component is dedicated to a given task, from low-level control to high-level processing. Components of the functional layer communicate with each other with the help of a software piece called the *middleware*. Two essential concepts are involved:

Data flow refers to the exchange of data between components. Data routing from one component to another is ensured by the middleware.

Control flow denotes calls to *services* that components typically provide to modify their behaviour. The availability of a component's service is also handled by the middleware.

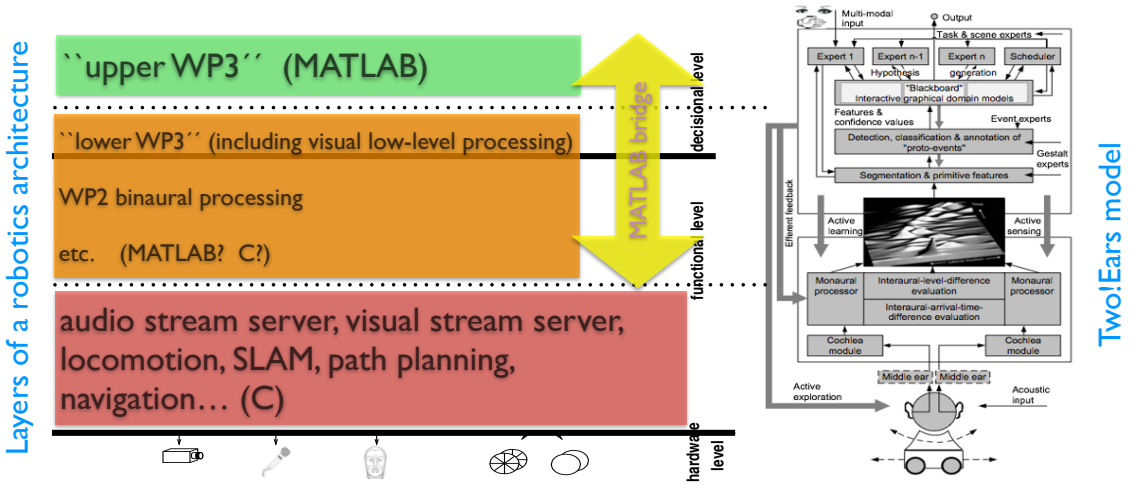


Figure 3.1: From the Two!Ears computational model (right) to a real time robotics software architecture (left).

Calls to services can be emitted by any other component of the functional layer—referred to as a Remote Procedure Call (RPC). They can also be emitted by a piece of software of the decisional layer (software monitoring the state of the robot, supervisor,...) or by a user by means of a generic interpreter. Figure 3.2 illustrates these concepts on a toy model.

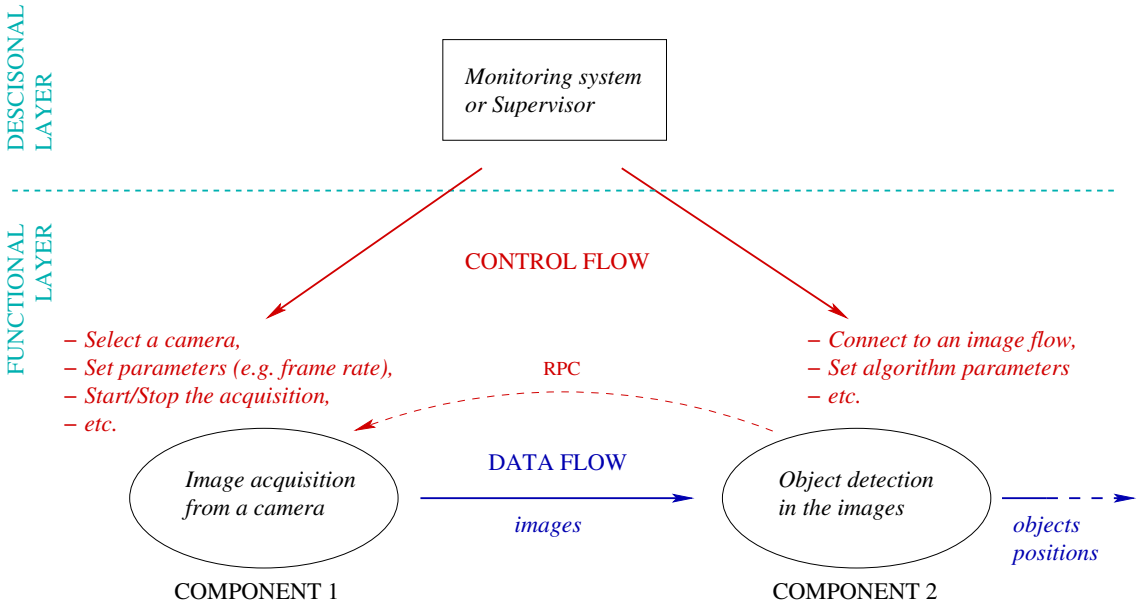


Figure 3.2: A simple component-based architecture to perform object detection in images. Two components are involved: one acquires images from a camera and streams them; the other runs an object detection algorithm. The data flow from the first component to the second one is shown in blue. Both components provide services that the decisional layer can call, shown in red. A Remote Procedure Call (RPC) is also illustrated here by a dashed red arrow.

Component-based software architectures offer great benefits in robotics (Brooks *et al.*, 2005), addressing typical issues such as modularity (so that the architecture can be distributed over a network of host machines), re-usability (common components can be used across robots without having to recode them), scalability, and even formal proofs of dependability.

3.1.3 *ROS*, a software platform for robotics

ROS (Robot Operating System) is a widely known software platform in robotics. It not only provides a middleware, but also implements a wide range of commonly-used functionalities into software components (such as localisation, mapping, path planning, obstacle avoidance, *etc.*), with a build system and a packaging system for easy compilation and installation. As claimed by the growing *ROS* community, *ROS* was built from the ground up to encourage open-source collaborative robotics software development. This makes *ROS* a common choice as a robotic software platform, as it is for TWO!EARS.

ROS embraces the principles of component-based software architectures, allowing concurrent and distributed computation, software reuse and rapid testing (O'Kane, 2013). The main *ROS* terminology is summarized here:

Nodes Software components using *ROS* middleware are called *ROS nodes*. They are independent processes running on one or several host machines.

Topics and messages Data flows are called *topics*. A node that outputs data *publishes* on a topic. A node that inputs data *subscribes* to a topic. The data elements flowing on topics are called *messages*. Each message is made of various data fields forming part of a data structure called *message type*. As a given topic only carries one message type, the term *topic type* is equally used.

Services and actions Nodes can provide *services* to control them. A service may take input parameters at its invocation, and may return output parameters upon its completion. Services that take a long time to execute (*e.g*, navigating along a planned path) are rather defined as *actions*, which provide feedback mechanisms during their execution.

Software in *ROS* is organized in *packages*. A package can contain *ROS* nodes, useful datasets, configuration files, etc. *ROS* packages are themselves organized into *stacks*, which are the primary mechanism in *ROS* for distributing software. For instance, the *ROS navigation* stack contains many packages dedicated to the navigation of a mobile base in a learnt map of its environment.

The officially supported platform for *ROS* is *GNU/Linux Ubuntu*. Different versions of *ROS* exist, with compatibility restrictions on *Ubuntu* versions[1]. Developments for TWO!EARS started using *ROS groovy* on *Ubuntu* 12.04 LTS, and now use *ROS indigo* on *Ubuntu* 14.04 LTS as the final version.

1 *cf. ROS* Enhancement Proposal 3: `http://www.ros.org/reps/rep-0003.html`.

3.1.4 *GenoM3*, a framework to design robotic components

The development of robotic components can be significantly improved by means of a tool called *GenoM3* (*Gen*erator *o*f *M*odules, version *3*)[2]. As a result of two decades of research on real time architectures for autonomous systems (Alami *et al.*, 1998)(Mallet *et al.*, 2010), *GenoM3* allows to develop robust, middleware-independent software components thanks to a model-driven approach, presented below on a toy example.

A *GenoM3* component is first defined by a description file, called the *dotgen* file, with the `.gen` extension. This file gathers in a single place all the definitions related to the component's interface, its control and data flows in particular. A typical installation of the *GenoM3* framework includes a toy component named `demo`, that controls the movement of a fictional robot along a single axis. A simple *dotgen* file for the `demo` component could look like this[3]:

```
component demo {

  struct demostate {
    double position; /* current position */
    double speed; /* current speed */
  };

  port out demostate Mobile;

  task motion {
    period 400 ms;
  };

  ids {
    demostate state;
  };

  activity GotoPosition (
              in double posRef = 0 : "Goto position in m") {
    doc    "Move to the given position";
    task   motion;
    codel <start> gpStartEngine() yield exec;
    codel <exec> gpGotoPosition(in posRef, inout state,
                        out Mobile) yield pause::exec, stop;
    codel <stop> gpStopEngine() yield ether;
  };
};
```

2 The *GenoM3* project is hosted at `https://git.openrobots.org/projects/genom3`.

3 The actual *dotgen* file of the `demo` component is more complete, deliberately simplified here for clarity purposes. It is available at `http://trac.laas.fr/git/demo-genom`.

On the basis of this *dotgen* model, *GenoM3* automatically generates real time code for tasks sequencing, middleware communication, etc. It also generates skeletons of functions that implement the algorithmic core run by the component. So, the developer just has to fill them with algorithms, written in separate *C* or *C++* source files, possibly linking to external libraries.

The syntax is explained in detail below.

Component definition

```
component demo {
  ...
};
```

This defines a component named `demo`. As a key feature, *GenoM3* allows to develop middleware-independent components, *i.e.* `demo` can be compiled for different middleware solutions without changing its source code. Middleware-dependent code is automatically generated by *GenoM3*. A clear separation of concerns between the algorithmic core and the middleware is thus conducted, helping towards an improved design of robotic components. Among the supported middleware solutions, *GenoM3* can create components for *ROS*. In this case, the built `demo` program is a genuine *ROS* node.

Data types

```
  struct demostate {
    double position; /* current position */
    double speed; /* current speed */
  };
```

Data types, such as the structure `demostate`, use a subset of the *OMG IDL* language. It allows generic definitions that can be shared between components.

Ports (data flows)

```
  port out demostate Mobile;
```

Ports are in charge of data flows coming in or out of the component (when using *ROS* middleware, they translate to *ROS* topics). Here, an output port named `Mobile`, exporting data of type `demostate`, is defined. It publishes the current position and speed of the fictional mobile robot to the external world.

Tasks

```
task motion {
  period 400 ms;
};
```

A *GenoM3* component can have multiple concurrent execution tasks. These tasks run the algorithmic core, made of atomic, non preemptable routines called *codels* (for "code elements", see the definition of services below). Here a periodic task `motion` is declared, to be further used in order to run a service `GotoPosition` to move the robot.

Internal Data Structure

```
ids {
  demostate state;
};
```

The `ids` tag defines the *Internal Data Structure* of the component. Memory sharing between concurrent tasks is safely handled by *GenoM3* through this data structure. The present *IDS* contains one field named `state` of type `demostate`. It internally holds the current speed and position of the robot at any time during execution.

Services (control flows)

```
activity GotoPosition (
           in double posRef = 0 : "Goto position in m") {
  doc    "Move to the given position";
  task   motion;

  codel <start> gpStartEngine() yield exec;
  codel <exec> gpGotoPosition(in posRef, inout state,
                    out Mobile) yield pause::exec, stop;
  codel <stop> gpStopEngine() yield ether;
};
```

Last, a *GenoM3* component can define services. They are either called *functions* for small operations which should be executed and finished almost instantaneously (similar to *ROS* services), or *activities* for operations that need time to perform (similar to *ROS* actions). A *function* consists of a single codel, while an *activity* is defined by a finite state machine, with one codel per state.

Here, an activity named `GotoPosition` is defined, run by the above task `motion`. It has one input parameter `posRef` that specifies the reference position to be reached by the

fictional robot. The state machine starts with a `start` codel, yielding to an `exec` codel that progressively moves the robot towards the reference position. During execution, the current position and speed are read from the *IDS* and exported on the output port. Once the reference is reached, the `stop` codel is executed. Transitions between codels happen at the period of the task, and the service can be interrupted during a transition.

To summarize

GenoM3 facilitates the development of essential features for robotic components, such as the definition of finite state automata with an optional clock as seen above, task concurrency and memory sharing between concurrent tasks. Other valuable properties were not illustrated in this example, such as clean interruption mechanisms and efficient error handling. Using the *GenoM3* framework results in highly robust, sustainable, reusable and middleware-independent robotic components. Though not used in TWO!EARS, *GenoM3* can also be coupled with formal validation and verification tools[4].

3.2 A software bridge between the deployment and development systems

3.2.1 Bridging *ROS* and *MATLAB*

As aforementioned, the TWO!EARS comprehensive software architecture includes an upper layer running *MATLAB* and a lower layer supported by *ROS* on a robotic platform. Therefore, a bridge between *MATLAB* and *ROS* is needed to handle control and data flows between the two layers. While being a popular need, the interface of *MATLAB* and *ROS* is a complex task giving rise to many usability concerns (Corke, 2015).

In January 2014, The MathWorks™ provided free *ROS* support through the *ROS I/O Package*, but with notable drawbacks such as the impossibility to call *ROS* services. So, during Year 1, a custom software bridge was designed at *CNRS*, suited to control and data exchange with *GenoM3* components by taking advantage of generic tools provided by *GenoM3*, exposed below in 3.2.2. In early 2015, The MathWorks™ released the *Robotics System Toolbox* for *MATLAB* R2015a or later, featuring better *ROS* support, and removed the former *ROS I/O Package* from its website.[5]. Meanwhile, in Year 2, the bridge from *CNRS* was fully revisited so as to cope with *GenoM3* as well as native *ROS*

4 BIP/D-Finder for instance, `http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html`. In collaboration with our colleagues from *CNRS* who designed *GenoM3*, the formal analysis of the *GenoM3*/*ROS* component specifically developed for binaural audio streaming is planned shortly after the end of the project, so that various temporal properties can be formally assessed.

5 *cf.* `http://www.mathworks.com/matlabcentral/answers/195837-why-am-i-not-able-to-find-the-ros-i-o-package-previously-available-on-matlab-central`.

components with improved performance. The final bridge and the *Robotics System Toolbox* are compared in Section 3.2.4.

3.2.2 Taking advantage of the *GenoM3* framework with the *genomix* server

The *GenoM3* framework, presented in Section 3.1.4, allows to develop robust, middleware-independent robotic components of a software architecture. It also comes with a set of useful tools to control them. In particular, *genomix*[6] is a generic server that can receive *HTTP* requests to call services and read data flows provided by *GenoM3* components, in a middleware-independent way. A *Tcl* client[7] of this server originally completes the *GenoM3* software suite. In Year 1, a similar client for *MATLAB* was developed, fully revised in Year 2 and released in open-source under the name *matlab-genomix*[8]. In addition, in Year 2, a server called *rosix*[9] was added to the suite in order to address any native *ROS* node in a generic way, using the same *HTTP* interface as *genomix*. With *matlab-genomix* as a client of *genomix* and *rosix* servers, this solution gives a complete interface between *MATLAB* and *GenoM3* or native *ROS* components, as illustrated in Figure 3.3.

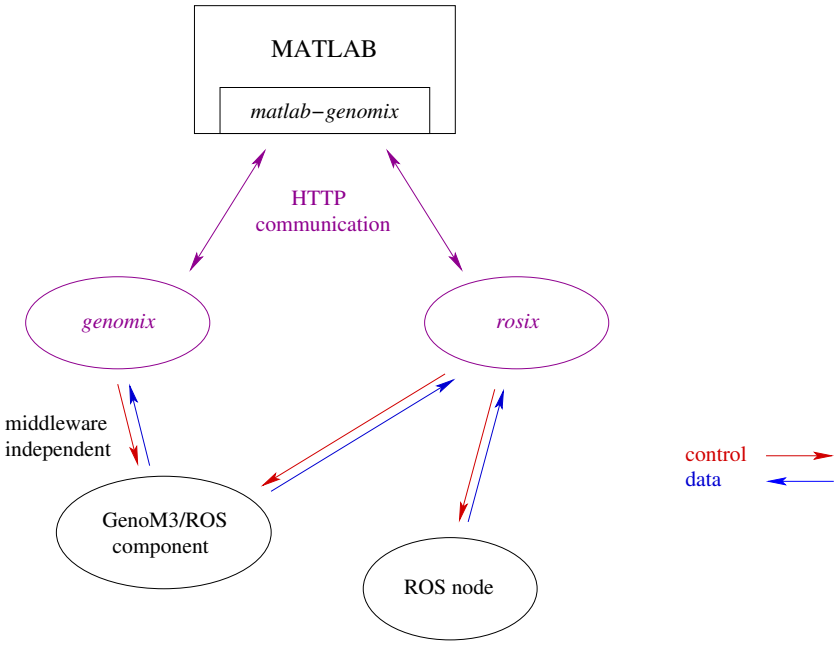


Figure 3.3: Use of *genomix* and *rosix* to bridge *MATLAB* and *ROS*. *genomix* allows to control *GenoM3* components and read their data flows independently of the middleware, while *rosix* can control and read data from any *ROS* node of the functional layer. *matlab-genomix* can be a client of any *genomix* or *rosix* server.

6 `https://git.openrobots.org/projects/genomix`
7 `https://git.openrobots.org/projects/tcl-genomix`
8 `https://git.openrobots.org/projects/matlab-genomix`
9 `https://git.openrobots.org/projects/rosix`

3.2.3 Sample use of the *matlab-genomix* software bridge

The *GenoM3* `demo` software component, introduced in Section 3.1.4, controls the movement of a fictional robot along a single axis. It provides a service named `GotoPosition` to move the robot to a target position, and an output port `Mobile` exporting the current position and speed of the robot. This section sketches the way how it can be controlled via the *matlab-genomix* bridge.

Let us assume that the `demo` component is running on a computer named `host-machine`, along with the *genomix* server. From any computer running *MATLAB*, a connection to *genomix* can be established with:

```
>> client = genomix.client('host-machine:8080');
```

A client handle is returned. It can be used to load the `demo` component in *MATLAB*:

```
>> demo = client.load('demo')

demo =
  component with properties:
              Stop: [function_handle]
          GetSpeed: [function_handle]
          SetSpeed: [function_handle]
            Mobile: [function_handle]
              kill: [function_handle]
      GotoPosition: [function_handle]
```

The returned handle holds methods specifically named after the services (*e.g.* `GotoPosition`) and ports (*e.g.* `Mobile`) that the `demo` component provides. *matlab-genomix* exploits *MATLAB* dynamic properties[10] to create these methods on-the-fly. Let us for instance call the `GotoPosition` service to move the fictional robot one meter ahead:

```
>> demo.GotoPosition(1.0);
```

This call is blocking, which means that the *MATLAB* prompt comes back only once the robot has reached the targeted position. This allows to simply wait for the completion of the service. A non-blocking call can also be made to perform other tasks in *MATLAB* while the service is running:

```
>> r = demo.GotoPosition('-a', 1.0);
```

This hands back the *MATLAB* prompt immediately. The service output status can be checked later using the returned request handle `r`. Last, let us see how to read data from the `Mobile` port:

10 *cf.* `https://www.mathworks.com/help/matlab/matlab_oop/dynamic-properties--adding-properties-to-an-instance.html`.

```
>> p = demo.Mobile();
>> p.Mobile
ans =
    position: 1.0
       speed: 0.0
```

The current robot position and speed are retrieved this way in *MATLAB*.

The above example only shows basic functionalities provided by *matlab-genomix*. The software comes with a more detailed tutorial[11].

3.2.4 Comparison between *matlab-genomix* and the *Robotics System Toolbox*

The *MATLAB* client of *genomix* and the *Robotics System Toolbox* are two prominent solutions to the integration of *ROS* features in *MATLAB*. Table 3.1 reports notable aspects on which the two solutions differ. In particular, the *HTTP* interface of *genomix* encapsulates data transferred from *ROS* to *MATLAB* into *JSON* objects. It has the benefit of providing high genericity, while the *Robotics System Toolbox* needs a separate software interface to load custom *ROS* data types in *MATLAB*. It however has the drawback of requiring extra processing to parse each *JSON* object in a *MATLAB* structure, but encoding and decoding are optimized to induce minimal overhead.

ROS support from the *Robotics System Toolbox*	*ROS* and *GenoM3* support from *matlab-genomix* with *genomix* and *rosix*
Proprietary and closed-source, developed by The MathWorks™.	Free and open-source, developed by *CNRS*.
For *MATLAB* ≥ R2015a.	For any *MATLAB* version.
Can publish data on *ROS* topics. *MATLAB* is considered to be a component of the software architecture.	Cannot publish data directly because *MATLAB* is seen as a supervisor. This limitation can be circumvented by requesting a component of the architecture to publish data through a service call.
Message types must be known *a priori*, custom messages are only possible through a separate interface.	*JSON* objects allow to de-serialize data structures in a highly generic way, without prior knowledge of their definition.
Strong data typing enables faster data transfer.	Data marshalling requires extra processing.
Solution for *ROS* middleware only.	Middleware-independent solution with *GenoM3* components, fully interfaced with *ROS* using the *rosix* server.

Table 3.1: Summary of differences between *matlab-genomix* and the *Robotics System Toolbox* for *ROS* support in *MATLAB*.

11 *cf.* `https://git.openrobots.org/projects/matlab-genomix/gollum/demo`.

3.3 Installation and license of the deployment software

3.3.1 Installation

The installation of the tools underlying the robotic software architecture is simple. The process is detailed in the TWO!EARS documentation (`docs.twoears.eu`).

- *ROS* is installed following the standard procedure[12] on *GNU/Linux Ubuntu*.
- *GenoM3*, *genomix*, *rosix* and *matlab-genomix*, are installed through the open-source compilation framework and packaging system *robotpkg*[13].
- *GenoM3* components developed for TWO!EARS are compiled from source (using the *Autotools*).

3.3.2 License

Most robotic software are released under permissive[14], BSD-like licenses. *ROS* core packages for instance use the BSD 3-Clause License. Unless otherwise stated, software from WP5 uses the BSD 2-Clause License[15].

Selecting a permissive license allows any other software piece to integrate or link to software from Work Package WP5 with minimum requirements. Other Work Packages can select a copyleft[16] license without any legal issue, as Work Package WP5 will not link to this software.

12 *cf.* `http://wiki.ros.org/indigo/Installation/Ubuntu`.
13 `http://robotpkg.openrobots.org/`.
14 A permissive license allows software to be redistributed with restricted access to the possibly modified code.
15 The license template is available at `http://opensource.org/licenses/BSD-2-Clause`.
16 A copyleft license requires that redistributed software remains free and open-source, and any modification or extension made to the software preserves the original rights.

4 The TWO!EARS deployment hardware and low-level software

This chapter surveys the design and implementation of physical test beds suited to the goals of TWO!EARS. Audio acquisition from a *KEMAR 45BB-2* Head-And-Torso Simulator (HATS) is first exposed (Section 4.1). Then, a controllable azimuth degrees-of-freedom (dof) and a stereoscopic sensor matched to the HATS are depicted (Sections 4.2–4.3). Further information is given on the mobile platforms supporting the HATSs of *CNRS* and *UPMC* (Section 4.4). Each hardware is described together with the companion low-level software (libraries, *ROS* nodes) necessary to its exploitation. Finally, off-the-shelf *ROS* stacks for Simultaneous Localization and Mapping (SLAM) and navigation are outlined (Section 4.5).

4.1 Audio acquisition from a binaural sensor

4.1.1 Hardware: the *KEMAR* Head-And-Torso Simulator

The anthropomorphic HATSs used for the project are *KEMAR Type 45BB-2* models, fitted with "Large" European-like ears[1]. One *G.R.A.S Type 26CS* microphone[2] is placed inside each ear (Figure 4.1). Associated to it is the amplifier, which ensures its supply from an external current source, and drives the audio signal to a Microdot output connector. Table 4.1 summarizes the specifications.

Downstream the microphones, Integrated Electronics Piezo Electric (IEPE) M28 Supply Modules[3] ensure the following functions:

- generation of a 4 mA constant current for the microphone supply;
- injection of the current into the cable and combination with the sound signal;
- extraction of the sound signal coming from the microphone out of the bias current;
- amplification, with a defined gain, and band-pass filtering of the audio signal.

1 See `http://www.gras.dk/45bb-2.html` and `http://www.gras.dk/media/docs/files/items/m/a/man_45BB_45BC.pdf`.
2 `http://www.gras.dk/26cs.html`.
3 `http://www.mmf.de/manual/m28mane.pdf`.

Figure 4.1: *G.R.A.S Type 26CS* Microphone

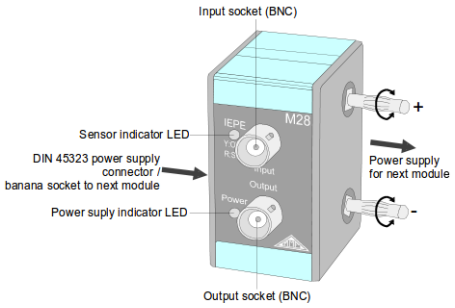


Figure 4.2: *IEPE Supply Module M28*

Specification	Value	Unit
Frequency Range	2.5 to 200 k	Hz
Slew Rate	20	V/μs
Input Impedance	20 // 0.4	GΩ // pF
Output Impedance	<50	Ω
Output Voltage Swing, min @ 24-28 V CPP voltage supply	8	Vp
Noise (A-Weighted) max	2.5	μV
Noise (A-Weighted) typical	1.5	μV
Noise (Linear 20Hz - 20kHz) max	6	μV
Noise (Linear 20Hz - 20kHz) typical	3.5	μV
Gain	-0.45	dB
Power Supply (Constant Current Power)	2 to 20 (typ. 4)	mA
DC bias voltage typical	12	V
Weight	3.0	g

Table 4.1: Specifications of *G.R.A.S Type 26CS* microphones

These IEPE devices comply with the IEEE 1451.1 standard for the output of piezoelectric transducers or microphones. Their outputs are fed to the used audio interface, namely, a *RME Babyface*, which can be accessed through ALSA[4] under *GNU/Linux*[5]. A software component named *BASS*, presented below, streams the acquired data to the *ROS* architecture.

4.1.2 Software: the *Binaural Audio Stream Server*

The *Binaural Audio Stream Server* (*BASS*) is a *GenoM3* component[6] in charge of acquiring binaural audio data from any ALSA-compliant hardware sound device, and of making it available to other components of the software architecture.

4 The Advanced Linux Sound Architecture (ALSA) is a part of the *Linux* kernel, providing drivers for audio devices, *cf.* `http://www.alsa-project.org`.

5 The *RME Babyface* has a *Class Compliant* mode for compatibility with standard USB audio devices, *cf.* `http://www.rme-audio.de/download/cc_mode_babyface_e.pdf`.

6 Available at `https://github.com/TWOEARS/audio-stream-server`.

BASS offers services to parameterize/start/stop the acquisition, and streams the captured data to an output port. In its capturing state, the sound device periodically delivers chunks of new data to the *BASS* component. Their size, commonly given in amount of frames[7], is set before starting the acquisition. *BASS* then pushes every new chunk to its output port, so that a sliding window of the most recent data is published. For instance, the port can be configured as a FIFO[8]-like buffer which contains the last two seconds of acquired signals.

BASS provides the following services.

- `ListDevices` lists ALSA sound cards available for acquisition. If the computer has multiple plugged sound cards, this allows to identify the card to use.
- `Acquire` starts the acquisition. It expects a few parameters which are: the identity of the sound device (retrieved with `ListDevices`); the sample rate; the size of chunks delivered by the sound device; and the size of the FIFO for the output port. The acquired samples are streamed on the output port, named `Audio`.
- `Stop` interrupts a running acquisition.

The example below shows how to get the audio stream from *BASS* in *MATLAB*, using the *matlab-genomix* bridge (section 3.2).

```
% Connect to genomix and load BASS
>> client = genomix.client('host-machine:8080');
>> bass = client.load('bass');

% Start the acquisition (values are just examples)
>> babyfaceID = 'hw:1,0'; sampleRate = 44100;
   chunkSize = 2048; portSize = 10;
>> r = bass.Acquire('-a', babyfaceID, sampleRate,
                    chunkSize, portSize);

% Check that starting acquisition was successful
>> if strcmp(r.status, 'error')
       error('Starting acquisition failed');
   end

% Get audio data in MATLAB
>> p = bass.Audio();
>> p.Audio
ans =
         sampleRate: 44100
      nChunksOnPort: 10
    nFramesPerChunk: 2048
```

7 Here, a frame is defined as a pair of left and right samples at a common sampling time.
8 First In, First Out (FIFO).

```
     lastFrameIndex: 6341580
               left: {1x20480 cell}
              right: {1x20480 cell}
              stamp: [1x1 struct]

% Stop the acquisition
>> bass.Stop();
```

For the needs of the TWO!EARS Summer School which took place in September 2015, *BASS* was deployed on a low-cost integrated binaural sensor. This sensor was made up with a polystyren spherical head, a pair of *ST Microelectronics MP34DT01* MEMS microphones[9] housed in custom 1/2-inch cylinders with suitable electronics, a *Logi PI* FPGA board[10] in charge of decoding the raw digital signals, together with a *Raspberry PI2*[11] computer running *GNU/Linux*, *ROS*, *GenoM3* and *BASS*. After ensuring that audio data could be streamed from an ALSA device connected to the *Raspberry PI2*, the genuine *BASS* was then interfaced with the *Logi PI* board. Details of this implementation were given in Deliverable D5.2@m24.

4.2 A *KEMAR* Head-And-Torso Simulator with a controllable azimuth degrees-of-freedom

Task 5.1 of TWO!EARS had initially planned to mount an anthropomorphic binaural head on a pan-tilt unit. Soon after the beginning of the project, the consortium decided to start from a complete *KEMAR* HATS and to endow its neck with a pan dof. The tilt dof was dismissed as it does not contribute enough to active motion. The characteristics of the hardware design and software libraries enabling the azimuth motion are hereafter described.

4.2.1 Hardware: devices for a controllable azimuthal degree-of-freedom

By default, the head of the *KEMAR* HATS is not rigidly linked to the torso, and can be moved manually in azimuth, with the possibility to lock it at some specified angles (Figure 4.3). The dark element of the assembly, endowed with an angle indicator, goes inside the torso. It constitutes the neck, as it remains visible between the torso and the head. The light grey item is rigidly attached to the head. The basic idea was to replace these two parts by an aluminium device designed on the basis of the CAD model of the *KEMAR* HATS (Figure 4.4). This device is screwed on the genuine mounting holes of the *KEMAR* torso, in exactly the same way as the original assembly mechanism (Figure 4.3).

9 http://www.st.com/web/en/resource/technical/document/datasheet/DM00039779.pdf
10 http://valentfx.com/logi-pi/
11 https://www.raspberrypi.org

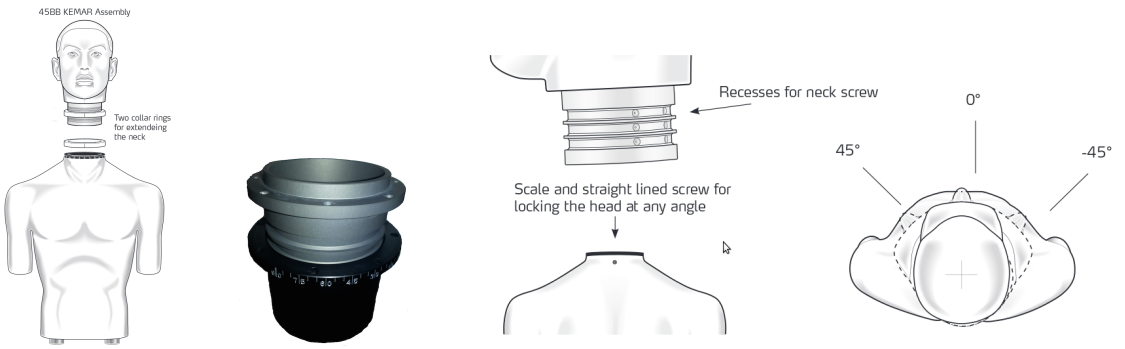


Figure 4.3: From left to right: assembly mechanism of the *KEMAR* HATS; angle indicator; neck, with locks at 0deg, 45deg, −45deg.

So, the integrity of the *KEMAR* HATS is ensured. Importantly, the device is endowed with holes so that the cables connected to the two microphones can transmit the binaural data to the audio interface through the lower part of the torso.

The servocontrol of the head is ensured by a set including a motor, its gearhead, an encoder, and an electronic controller. A brushless DC motor was selected, as it constitutes a good compromise in termes of compactness, simplicity, audible noise, and maintenance. Its coupling with a planetary gearhead enables a sufficient output torque at startup. A relative quadrature encoder was integrated at the output of its shaft to measure the azimuth of the head with a resolution of 73728 pulses per turn.

An *Harmonica Controller* from *ELMO* completes this set. This "Compact and Smart Digital Servo Drive" integrates all the processing and power switching elements necessary to drive the motor in the considered embedded context. Several feedback control options are provided, such as position or velocity setpoints, etc. Communication with the outside world is ensured through the standard CAN bus protocol.

In the first design, head rotation limits were detected by two magnetic Hall effect sensors mounted on the static part of the mechanism and connected to the *Harmonica* motor controller. A magnet stuck on the rotating part triggered a response when it induced a strong enough magnetic field, *i.e.*, when it approximately faced one sensor. To improve repeatability, during Year 3, this solution was traded for photoelectric proximity sensors placed in the static part, and for a reflective element sticked under the moving part (Figure 4.4-left). Each sensor emits a focalized light beam. On the basis of the sensed return paths, custom logical functions implement the detection of left and right end-of-course positions. This new design is implemented on a PCB. It leads to a higher accuracy and repeatability, and increases the admissible range of the head azimuths from [−80deg; +80deg] to [−90deg; +90deg]. Note that negative logic is used, so that if a sensor gets disconnected, head movements are immediately stopped to prevent any damage.

The whole set (*Harmonica Controller*, IEPE modules, PCB, ...) has been rewired and safely housed into the *KEMAR* HATS. The two DIN rails holding these elements are fixed on the lower plate (Figure 4.4). So, users have a limited access to the components and do

not need to disconnect or reconnect cables anymore when the HATS must be moved. On its rear, four connectors are now available:

- A *DB-9* connector to a *standard CAN bus protocol* for communication and commands;
- Two *BNC* plugs for microphones;
- One *LEMO* plug for 24 V-DC input power; this voltage is available on the *Jido* and *Odi* platforms from *CNRS* and *UPMC*, and can be easily provided by an external power supply for a standalone usage of the motorized HATS.

Detailed assembly instructions for the mechanical and electrical designs are available in the appendix (Section 7.2).

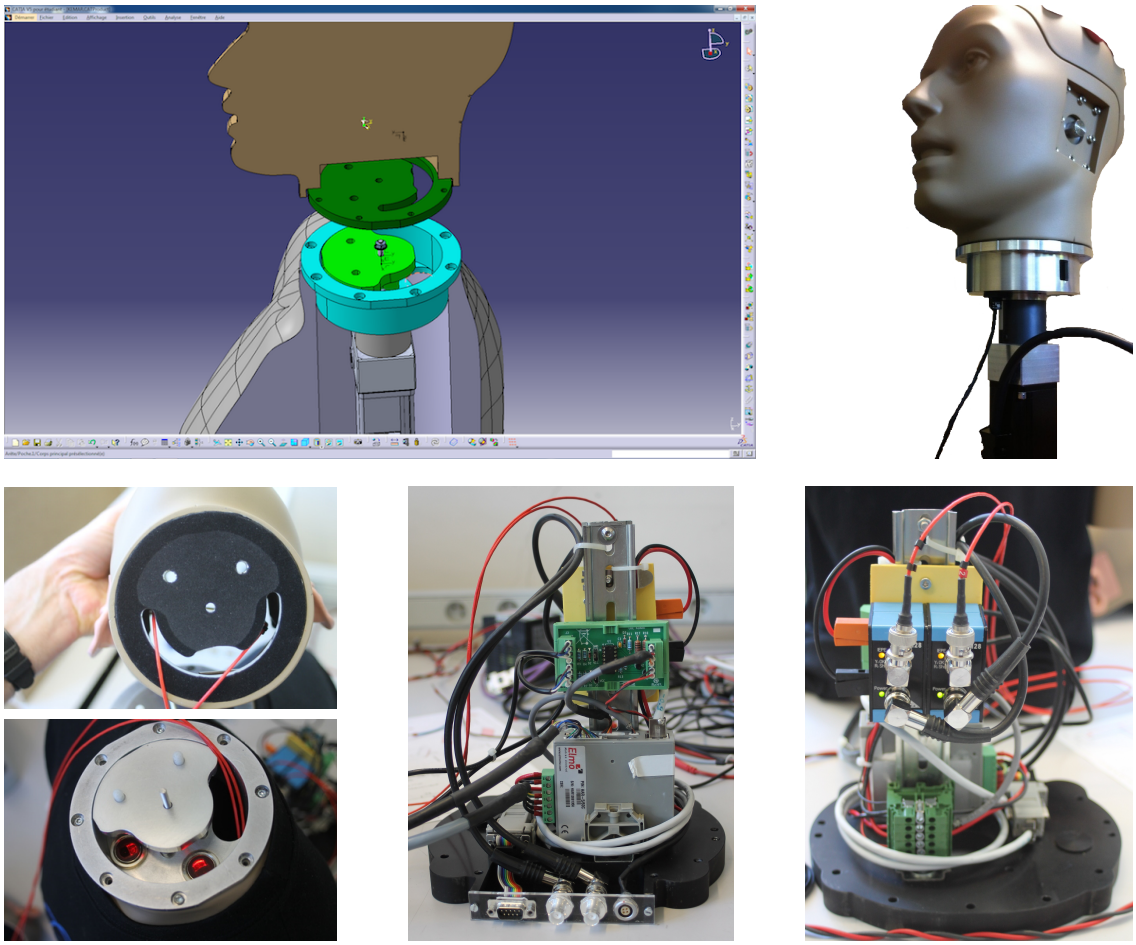


Figure 4.4: Top: CAD design (left) and implementation (right) of the *KEMAR* motorization system. Bottom-left: Aluminium parts to be fixed under the head and inside the torso. Bottom-middle and Bottom-right: packaging of the full system with IEPE modules (blue), *Harmonica controller* (grey), photoelectric sensors logic PCB (green) and coaxial microphones cables (red). Note the new HATS rear connectors, from left to right: one *DB-9* connector for *CAN bus protocol*, two *BNC* plugs for microphones, one *LEMO* power supply connector.

4.2.2 Sofware: low-level libraries and *GenoM3* component

Three different custom, open-source low-level libraries[12] have been developed for the motorization of the *KEMAR* head.

- `Socketcan` provides an interface with the *GNU/Linux* socket CAN layer. Its main goals are to initialize or end the communication with the CAN bus controller, and to send or receive messages.
- On the top of it, `Harmonica` provides an interface with the *ELMO Harmonica Motor Controller* which drives the motor. It includes functions to initialize and stop this controller, as well as to set and get the motor position through it.
- On the top of them, `Kemar` provides an interface specific to the *KEMAR* HATS itself. It includes functions for head homing, position control or velocity control. These entail the aforementioned limit sensors.

The `Kemar` library is encapsulated into the `kemar` *GenoM3/ROS* module[13] of the TWO!EARS deployment system so as to enable its concurrent execution with other tasks. It was initially designed during Year 1, and updates were brought all along the project, especially after the hardware upgrade. The component provides an output port filled with the current state of the motor, *i.e.*, its current position and velocity, and the following services are offered.

- `Homing` must be called at least once prior to other services. It first turns the head towards left, then right, until it reaches the limit sensors. This calibrates its position encoder and deduce the maximum left and right admissible rotations. Then, the head is set to the middle position, defined as the origin (0 deg) for position control.
- `SetVelocity` sets the reference velocity (in $\mathrm{deg.s^{-1}}$) during the position control of the head (services `MoveAbsolutePosition` or `MoveRelativePosition` listed below). The default value, defined after homing, is $100\,\mathrm{deg.s^{-1}}$.
- `MoveAbsolutePosition` is a service for position contol. It drives the head to a setpoint defined with respect to the origin.
- `MoveRelativePosition` is similar to `MoveAbsolutePosition`, except that it moves the head relatively to its current position.
- `ControlInSpeed` is a service that moves the head at a given constant velocity (in $\mathrm{deg.s^{-1}}$) until it is called again (*e.g.*, at $0\,\mathrm{deg.s^{-1}}$ to stop the head) or until the left or right limit value is reached. This velocity is independent from the reference velocity used during position control.

The example below shows a simple position control from *MATLAB*, using the *matlab-*

12 Available at `https://git.openrobots.org/projects/elmo-axis-libs`.
13 Available at `https://github.com/TWOEARS/kemar-control`.

genomix bridge (section 3.2).

```
% Connect to genomix and load the kemar component
>> client = genomix.client('host-machine:8080');
>> kemar = client.load('kemar');

% Execute the homing procedure
>> kemar.Homing();

% Set the reference velocity for position control to 50 deg/s
>> kemar.SetVelocity(50);

% Move the head to 45 deg with a blocking call
>> kemar.MoveAbsolutePosition(45);
% The prompt returns once the head has reached the position
```

4.3 Incorporation of stereovision on the *KEMAR* HATS

This section describes the final implementation of a human-like stereoscopic system suited to any *KEMAR Type 45BB-2* HATS. This non-intrusive sensor consists in micro-cameras mounted on 3D-printed glasses (Figure 4.5). Visual functions suited to this design were implemented, for human detection and tracking as well as object detection and localization (Sections 6.2–6.3).

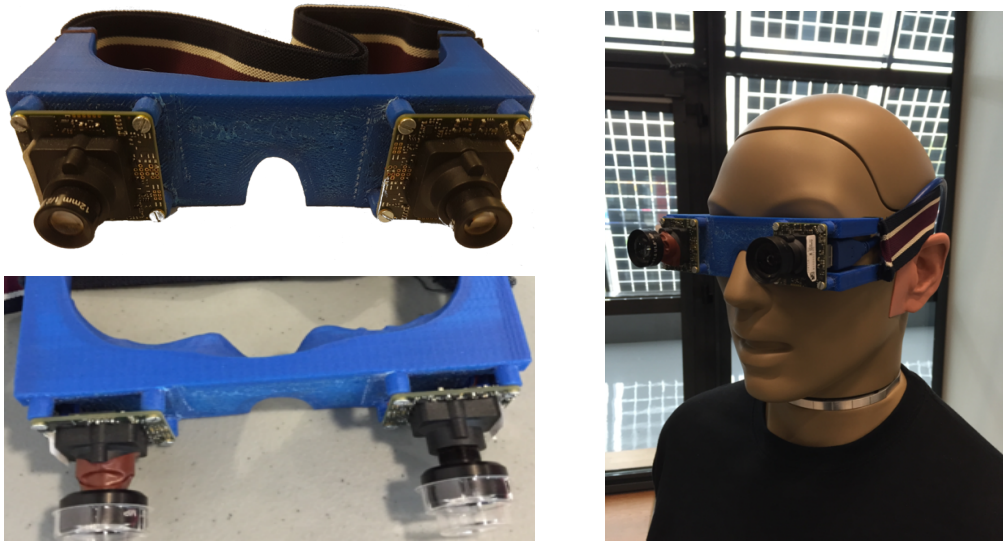


Figure 4.5: Custom 3D-printed glasses: (left) with two types of lenses; (right) mounting on the *KEMAR* HATS.

4.3.1 Hardware: lenses, sensors and 3D-printed glasses

On the basis of the *KEMAR* CAD model, *CNRS* designed a pair of glasses which steadily fits at the level of the eyes. On its 10 cm-baseline (enlarged w.r.t. the *KEMAR* eyes for improved stereovision performance) it incorporates a pair of *IDS* UI-3241LE-C-HQ μeye micro-cameras[14] as well as their wires and connectors. These are endowed with a $1/1.8''$ CMOS sensor featuring a 1280×1024 matrix of $5.3\,\mu$m pixels using a global shutter, and USB 3.0 interface. Their size and weight are $36.0 \times 36.3 \times 20.2$ mm and 12 g.

A comprehensive Application Programming Interface (API) is provided by *IDS*[15], to configure the devices and retrieve images. For accurate stereoscopic reconstruction, the left and right images are synchronized. One camera, considered as the master, generates the hardware trigger signal for the slave one.

To obtain high 3D accuracy after triangulation when tracking humans, Lensagon B16020S12 and Lensagon BSM12016S12 lenses were first mounted on the glasses. Their respective weights are 4.2 g and 6 g. Their 16 mm and 12 mm focal length led to a high angular precision though at the expense of a reduced field of view (22 deg or 31 deg, respectively). 3D localization accuracy results were satisfying. Methods for object detection and segmentation based on dense RGB-D point clouds also worked well using depth maps generated by stereovision. However, due to the restricted stereovision field of view, a rotation of the head had to be performed in order to scan the scene and determine areas of interest. Consequently, Lensagon BM5518S12ND were installed. Their 5.5 mm focal length leads to a wider field of view (64.5 deg), but implies a heavier weight (17 g). In addition, the angular resolution drops, and so does the 3D accuracy.

4.3.2 Software: acquisition, calibration, rectification and triangulation

Acquisition

The *ROS* open-source *ueye_cam*[16] package performs image acquisition using functions from the aforementioned API. The output are two *ROS* topics with images from both cameras, synchronized and equally timestamped.

Calibration

Monocular calibration is the process of estimating the *intrinsic parameters* (focal length, image parameters and principal point) as well as the *extrinsic parameters* (translation and rotation turning the 3D world coordinates to 3D camera coordinates), see Figure 4.6.

14 `https://en.ids-imaging.com/store/ui-3241le.html`
15 `https://en.ids-imaging.com/manuals/uEye_SDK/EN/uEye_Manual/index.html?c_programmierung.html`
16 `http://wiki.ros.org/ueye_cam`

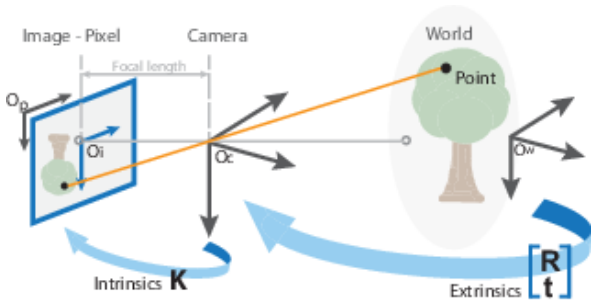


Figure 4.6: Representation of *intrinsic* and *extrinsic* parameters (from `http://mathworks.com/help/vision/ug/camera-calibration.html`).

This is performed by the *camera_calibration ROS* package[17], which supports monocular and stereoscopic calibration. Stereoscopic calibration involves the determination of the two sets of camera parameters plus the relative translation between the two camera frames. The underlying algorithm comes from the *OpenCV* library[18], on the basis of the so-called Plumb Bob model (pinhole + optical radial and tangential distortions) which projects the 3D scene onto the image plane using perspective transformation[19]. The intrinsic and extrinsic parameters are obtained by moving a calibration pattern (with easy-to-extract feature points, *e.g.*, a chessboard) to various positions and orientations in front of the cameras.

Rectification

The *stereo_image_proc ROS* package performs stereoscopic image rectification[20]. The raw images are projected on a common plane in such a way that each pair of epipolar lines, *i.e.*, each pair of intersections of the image planes with the plane passing through the optical centers and a 3D point, lies on a single row, see Figure 4.7. These rectified images are used for the evaluation of the stereoscopic system in Section 4.3.3, for human detection (Section 6.2) as well as object detection (Section 6.3).

Besides this, a disparity map which encodes the difference in horizontal coordinates of corresponding image points and a 3D point cloud are obtained and published (Figure 4.8).

Triangulation

Triangulation is the process enabling the recovery of the world coordinates of a 3D point from the 2D image coordinates of its projections on left and right images. This is a common problem of computer vision (Daniilidis and Eklundh, 2008).

17 `http://wiki.ros.org/camera_calibration`
18 `http://opencv.org/`
19 `http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`
20 `http://wiki.ros.org/stereo_image_proc` – `http://wiki.ros.org/image_pipeline/CameraInfo`

Figure 4.7: Pair of raw images (left) and pair of rectified images (right) with two epipolar lines, from `http://wiki.ros.org/stereo_image_proc`.

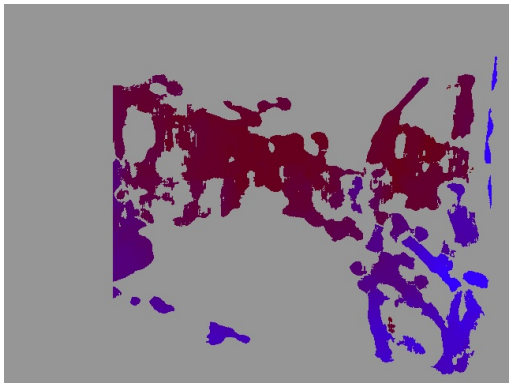


Figure 4.8: Disparity map, from `http://wiki.ros.org/stereo_image_proc`.

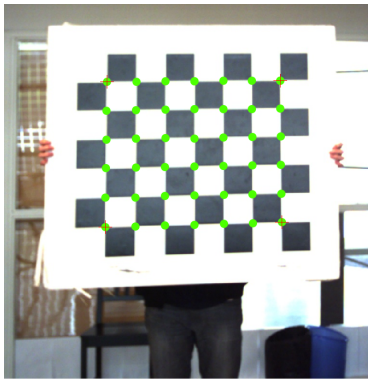


Figure 4.9: Automatic corner extraction in a chessboard-like calibration pattern.

4.3.3 Evaluation of the stereoscopic system

The accuracy of the stereoscopic system was evaluated. A chessboard-like calibration pattern was positioned fronto-parallel with the image plane of the camera at five distances (1m, 2m, 3m, 4m, 5m) corresponding to the useful range. The projections of the corners of the squares were extracted and stereo matched in the left and right images. The 3D positions of these corners were then estimated through the stereoscopic reconstruction process. For each of the five distances, a plane was fitted over these 3D points. This plane was then used as a reference against which the distribution of the computed depths is assessed. Figures 4.10 and 4.11 show the results for the reference distances 1 m and 5 m, respectively.

The curves plotted in Figure 4.12 represent the means of the estimates of the distances between the 3D points and their projection, as a function of the genuine distances. The red one, obtained after doing the calibration with a 21.0×29.7 cm chessboard, with 29 mm side squares, shows that the obtained model is suited for ranges below 2.5 m. Estimated distances were much less accurate beyond that limit because corners could be hardly extracted.

To get an accurate model, the points extracted from the calibration pattern during the calibration should be at distances where the system is expected to perform. Therefore, distances around 5 m should be taken into account, which was not the case for the red

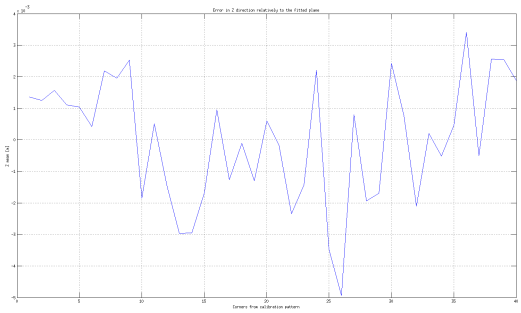


Figure 4.10: Error along the Z direction relatively to the fitted plane *vs* index of the corners extracted on the calibration pattern, for the distance 1 m. Total vertical range: $[-0.005\text{m}; +0.004\text{m}]$; step between tick marks: 0.001m.

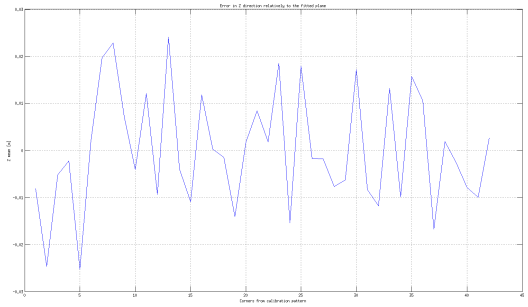


Figure 4.11: Error along the Z direction relatively to the fitted plane *vs* index of the corners extracted on the calibration pattern, for the distance 5 m. Total vertical range: $[-0.03\text{m}; +0.03\text{m}]$); step between tick marks: 0.01m.

curve in Figure 4.12. The calibration algorithm included in the *camera_calibration ROS* package allows the use of multiple calibration patterns. Therefore, for a successful corner extraction, an additional 95×105 cm chessboard, with 108 mm sided squares, was used at large distances (Figure 4.9). The red curve of Figure 4.12 clearly diverges from the ground truth dotted case for depths above 3 m, while the blue one stays quite close to the ground truth. The maximum distance between the fitted plane and the triangulated 3D point that is located farthest from it is plotted in Figure 4.13. At the distance of 5 m, the maximum reported error is 2.5 cm using the two-pattern calibrations, and this measurement is coherent with the predicted acurracy of the stereovision system. For the same ground truth distance of 5 m, the computed depth using the single-pattern calibration is 4.1 m, and the distance between the fitted plane and the farthest point is 4.6 cm. These results confirm the relevance of using two different sizes of calibration patterns to achieve accurate 3D reconstruction.

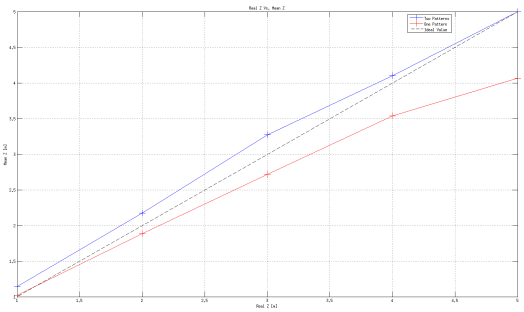


Figure 4.12: Means of the depth estimates computed by stereoscopic reconstruction *vs* genuine depth, for: (red) one-pattern calibration; (blue) two-pattern calibration. Total horizontal range: $[1\text{m}; 5\text{m}]$; total vertical range: $[1\text{m}; 5\text{m}]$; step between tick marks: 0.5m.

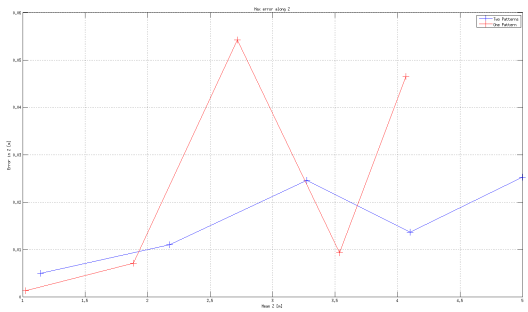


Figure 4.13: Maximum error along the Z-axis for: (red) one-pattern calibration; (blue) two-pattern calibration. Total horizontal range: $[1\text{m}; 5\text{m}]$; total vertical range: $[0\text{m}; 0.06\text{m}]$); step between tick marks: 0.01m.

4.4 The mobile platforms *Jido* (@CNRS) and *Odi* (@ISIR)

4.4.1 Robot at CNRS: *Jido*

A Hardware

At *CNRS*, *Jido* is based on a *Neobotix* MP-L655 mobile platform[21]. Its is robust, silent, and its payload is about 60 kg. It has been partially re-wired, its power system (distribution board, converters,...) has been updated, its caster wheels have been replaced. An Intel® Core™ i7 CPU E610 @2.53 GHz / 4 GB RAM fanless computer has been inserted. USB 2.0 ports, together with power lines delivering 5V, 12V and 24V voltages, are available to extend its hardware. A mechanical adaptater has been designed and installed to carry the *KEMAR* HATS. One Control Area Network (CAN) bus interface is dedicated to locomotion, and another one connects the motorization system of the *KEMAR* HATS. The driving motors and their relative encoders are connected to Harmonica controllers[22] similar to the one used for the motor of the neck of the HATS. The maximum translation and rotation velocities are $0.8\,\mathrm{m.s^{-1}}$ and $1\,\mathrm{rad.s^{-1}}$. The embedded sensors consist of a fairly accurate odometry (due to reduced slipping and high encoders resolution), and two SICK LMS200 laser range finders on the front and back sides of the robot.

B Software: *ROS* stack for *Jido* locomotion

Jido's base computer runs *ROS indigo* on *GNU/Linux Ubuntu* 14.04 LTS. A custom *ROS* stack named *jido_ros* was developed for low-level perception and control. It features the

21 `http://www.rcs.hu/roboshop/Neobotix/MP-L655_A.htm`
22 `http://www.elmomc.com/products/harmonica-main.htm`

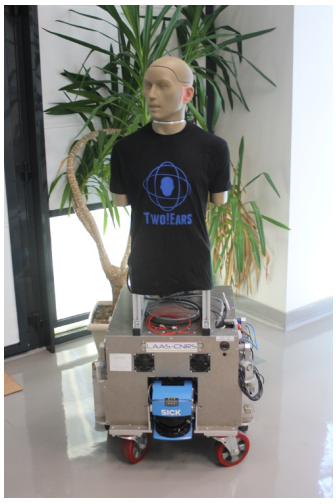


Figure 4.14: *Jido* @*CNRS*

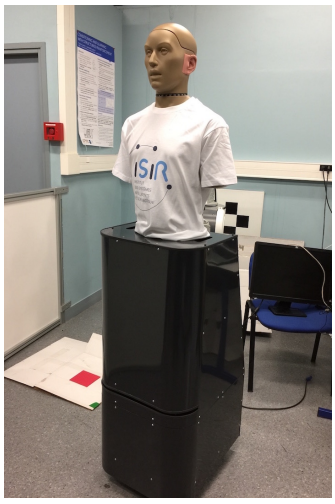


Figure 4.15: *Odi* @*UPMC*

following packages, which also comply with the standard interface of the *ROS navigation* stack, so that the platform can be controlled in a highly generic way.

- *jido_description* includes an XML representation of the robot's model in the Unified Robot Description Format (URDF).
- *jloco* provides a node to drive the wheels. It reads velocity commands from a *ROS* topic, and publishes the robot odometry on another one.
- The main package *jido_base* provides files which can quickly bring the base into operation, by launching nodes provided by other packages either from *jido_ros* or standard off-the-shelf *ROS* stacks. These nodes load the drivers for the laser rangefinders.
- *jido_teleop* provides a node that receives inputs from a joystick and turns them into velocity commands. It publishes the commands on the topic subscribed by the *jloco* node, letting the joystick control the motion of the base.
- *jido_2dnav* handles navigation by instanciating *ROS* nodes from the *ROS navigation* stack (described in Section 4.5) for localisation, path-planning and obstacle avoidance. This package also sets navigation parameters that fit the characteristics of the *Jido* platform and its environment.

4.4.2 Robot at ISIR: *Odi*

A Hardware

UPMC purchased (on its own funds) the mobile robot *Odi*, specifically designed for TWO!EARS by Enova ROBOTICS[23]. *Odi* was delivered on January 2016. Its maximum payload is 20 kg. *Odi* embeds an Intel® Core™ i5-4300U CPU @1.9 GHz / 8 GB *DDR3 RAM* computer, and CAN bus interfaces. Its maximum translation and rotational velocity are about $1\,\mathrm{m.s^{-1}}$ and $0.5\,\mathrm{rad.s^{-1}}$. In addition to proprioceptive information provided by wheels encoders, *Odi* is endowed with a Light Detection And Ranging (LIDAR) sensor enabling range estimation up to 5 m with about 1 % accuracy. Ethernet and USB 3.0 ports, together with embedded power lines delivering 5V, 12V and 24V voltages, are available to extend the robot hardware. Since September 2016, *Odi* carries the *KEMAR* HATS of *UPMC* equipped with the neck motorization system designed and manufactured at *CNRS*. A detailed user manuel of the system is available in the Appendix section. It provides all the information needed to proceed with a fresh *Odi* installation, from scratch, for both the hardware and software aspects.

23 `https://www.enovarobotics.eu/`.

B Software: *ROS* stack for *Odi* locomotion

Odi also runs *ROS indigo* on *GNU/Linux Ubuntu* 14.04 LTS. A specific *ROS* locomotion stack has been developed, which features the following packages.

- *kemar_robot* includes everything needed to handle motor commands and laser startup.
- *roboteq_driver* is the actual driver used by *ROS* to control the two motors of *Odi*.
- *odometry_listener* is in charge of *Odi* odometry.
- *kemar_teleop* allows to teleoperate the platform through a keyboard or joystick.
- *kemar_navigation* is in charge of *Odi* navigation, and includes everything needed to perform environment mapping and/or navigation inside (this includes the geometric parameters of the robot itself, i.e. wheel spacing and diameters, etc.).

These packages comply with the standard interface of the *ROS navigation* stack, so that all the developments on *Jido* can be easily ported to *Odi* without any change in the code.

4.4.3 Condition for an omnidirectional head

Jido and *Odi* are non-holonomic differential wheeled bases. The nonholonomic constraint imposes that they can only move tangentially to their trajectory, *i.e.*, any instantaneous motion pointing towards the axis of their wheels is not allowed. In other words, the degrees of freedom of each of them are the tangent linear velocity vector v_{base} and the angular velocity ω_{base} around the vertical axis. As a first approximation, the broadside direction of the *KEMAR* head is supported by v_{base}, and the rotation axis of the *KEMAR* neck is vertical, belongs to the midperpendicular plane of the wheels, and situated at a distance D ahead from the midpoint of the wheels. Its angular position and velocity with respect to the rigidly linked torso and robot basis are respectively denoted by q and ω_{head}.

As explained in Deliverable D5.2@m24, as soon as $D \neq 0$, it is possible to set the velocities of the *KEMAR* head to any 3-tuple v_y, v_z, ω_x, with v_z,v_y its linear velocities along the broadside (front) and interaural (left) directions and ω_x its angular velocity around the vertical axis. The three corresponding control inputs $v_{\text{base}}, \omega_{\text{base}}, \omega_{\text{head}}$ are recalled below (Cadenat, 1999):

$$\begin{bmatrix} v_{\text{base}} \\ \omega_{\text{base}} \\ \omega_{\text{head}} \end{bmatrix} = \begin{bmatrix} -\sin q & D\cos q & 0 \\ \cos q & D\sin q & 0 \\ 0 & -1 & -1 \end{bmatrix}^{-1} \begin{bmatrix} v_y \\ v_z \\ \omega_x \end{bmatrix}. \tag{4.1}$$

In practice, these equations hold up to the limiting values of $v_{\text{base}}, \omega_{\text{base}}, \omega_{\text{head}}$.

4.5 Off-the-shelf *ROS* stacks for SLAM and navigation

The *ROS* low-level components specific to each robot (locomotion, sensor handling) have a standard interface. So, on the top of them, platform-independent components can be used, what enables reproducible research. These can consist in specifically developed *GenoM3/ROS* modules, or in off-the-shelf solutions such as publicly available *ROS* stacks. Within this last category, some stable and widespread packages available under open-source license have been selected for map building and autonomous navigation.

To move around safely and autonomously, a robot must be able to localize itself in the two-dimensional (2D) environment and navigate between any set of location coordinates (x, y, θ). This implies the planning of a reference path in free space complying with its kinematic contraints, as well as the reflexive execution of this path while avoiding unexpected or moving obstacles (people, etc.). The off-the-shelf celebrated *ROS navigation* stack[24] has been used so far, which gathers some popular achievements. It relies on a map, to be built beforehand by an algorithm called Simultaneous Localization and Mapping (SLAM). Besides, the important frames of the robot (base frame, position and attitude of all the exteroceptive sensors with respect to this frame,...) must be described in the Unified Robot Description Format (URDF)[25]. They are published by the *ROS* node *robot_state_publisher*[26] as a tree in the *ROS* package *tf*, in charge of maintaining a tree of frames over time and compute changes of coordinates.

Figure 4.16 shows a typical set of components concurrently running on *Jido* and *Odi* during an elementary navigation task. Platform-specific and platform-independent components can be easily distinguished.

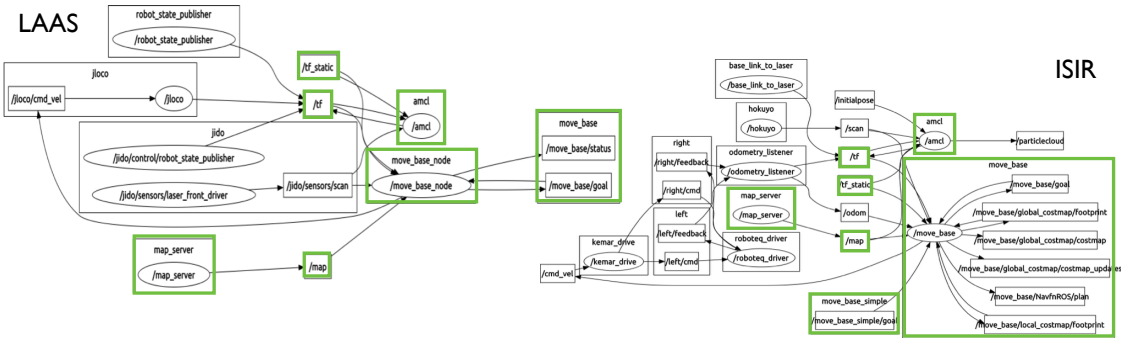


Figure 4.16: Interacting components in sample experiments with *Jido* or *Odi*. *ROS* nodes and topics are respectively represented by ellipses and boxes. Bigger boxes that surround nodes and topics are namespaces that gather closely related resources. Parts surrounded in green, from the *ROS navigation* stack (see Section 4.5.2), are platform-independent.

24 `http://wiki.ros.org/navigation`
25 `http://wiki.ros.org/urdf`
26 `http://wiki.ros.org/robot_state_publisher`

4.5.1 Map building

A popular SLAM algorithm for map building is implemented in the off-the-shelf *gmapping*[27] *ROS* package, included in a dedicated *ROS* stack. It is a *ROS* wrapper for OpenSlam Gmapping[28]. It features a Rao-Blackwellized particle filter assimilating laser and odometer measurements and combining them with the movement of the robot (Grisetti *et al.*, 2007). The robot must be driven throughout the environment, *e.g.*, manually by using the joystick. The map in progress can be viewed in real time on the 3D visualization tool *rviz*[29]. Once the user deems the map satisfactory, he/she stores it in a file for further use in localization and navigation components.

gmapping requires a fine tuning of some parameters (out of a total of 36), such as the maximum range of the laser sensor, the dynamic noise of the prior motion model, the number of beams to skip in each scan to reduce computation time, the number of particles and the resolution of the map (meters per grid).

4.5.2 Navigation

The *ROS navigation* stack can be broken into three main packages.

- *amcl* performs odometry and laser based localization.
- *map_server* broadcasts map data on the *ROS* topic */map*.
- *move_base* features a global planner to compute an admissible path between two locations, and a real time, reflexive, execution of such a path which includes the detection of unexpected obstacles as well as local motion strategies to avoid them.

The *amcl*[30] *ROS* package (for Adaptive Monte Carlo Localization) considers a robot moving in a 2D environment mapped with *gmapping*. It implements a stochastic estimation of the pose of the robot into this map by means of a particle filter, on the basis of the time record of odometer and laser measurements. Three categories of parameters are entailed in the configuration of *amcl*:

- filter tuning (minimum and maximum allowed number of particles,...);
- laser model (minimum and maximum scan range,...);
- odometry model (differential *vs* omnidirectional, etc.)

Two 2D maps, named *costmaps*, maintain information about where the robot can navigate in the form of an occupancy grid (each cell of which is associated with an occupancy

27 `http://wiki.ros.org/gmapping`
28 `http://openslam.org/gmapping.html`
29 `http://wiki.ros.org/rviz`
30 `http://wiki.ros.org/amcl`

probability by an obstacle). One is meant for global planning, by creating long-range plans over the entire environment. The other one is used for local motion and obstacle avoidance. Therefore, four groups of parameters need to be configured.

- *Common costmap parameters*: thresholds on obstacle information (e.g. range) and on the footprint of the robot; etc.
- *Global costmap parameters*: coordinate frame of the global costmap; reference frame of the robot base; update frequency of the costmap; etc.
- *Local costmap parameters*: coordinate frame of the local costmap; reference frame of the robot base; size (width and height in meters) and resolution (meters/cell) of the costmap grid; etc.
- *Motion parameters*: admissible maximum velocity of the mobile base; tolerances to reach the goal; etc. given a plan and a costmap.

move_base executes the following steps to safely navigate to a location depicted by a (x, y, θ) coordinate tuple in the world frame. First, the robot is localized in the environment map, executing if necessary a "recovery behavior" (360° rotation around its vertical axis). Then, the global planner computes the admissible shortest path from the current location to the goal. A trajectory controller ensures the execution of this path. It entails a real time laser based obstacle detection algorithm, and brings local changes to the planned path if necessary. In case of failure, the robot can even move backwards.

The *rviz* graphical visualization interface can also be used, to define a goal in an intuitive way. Internally, it publishes such a tuple on a specific *ROS* topic from the *ROS navigation* stack called *move_base/goal*. For the TWO!EARS project, the goal situations of the robot must be often defined at the *MATLAB* level. Therefore, a *GenoM3* component, named *sendPosition*, has been coded, which provides the user with the possibility to control the robot either in *absolute* mode (*i.e.*, with respect to the world frame) or in *relative* mode (*i.e.*, with respect to its current location).

5 Bringing the Auditory Front-End into the ROS architecture

The role of the Auditory Front-End (AFE) is to turn acquired auditory signals—provided from the functional layer by the *Binaural Audio Stream Server* (*BASS*)—into higher level features exploited by the decisional layer, see Deliverable D2.2@m12. Its original implementation was carried out under *MATLAB*. However, it was not developed with real time constraints in mind: no concurrency was available between processors, and guaranteed computation time could hardly be satisfied when extracting a lot of features. As already argued in Deliverable 5.2@m24, it has then been decided to implement a new AFE right at the functional layer (*cf.* Section 3.1), supported by *ROS*.

This chapter precisely describes this *ROS* implementation. The *GenoM3* framework has been used, as it eases the specification, development and tests. The algorithmic core of the implementation, consisting in a standalone library called *openAFE*, is first described (Section 5.1). On this basis, a new *ROS* node, named *rosAFE*, has been implemented. Its description is proposed in Section 5.2, highlighting concurrency aspects. Finally, the way how each processor is actually instantiated by a supervisor is described in Section 5.3. This supervisor has been written in *MATLAB*, thus allowing a smooth transition between the *MATLAB* and *ROS* implementation of the AFE.

5.1 *C/C++* implementation of the AFE algorithmic core: the *openAFE* library

The migration from the genuine AFE to a *ROS* implementation requires the transcoding of the algorithmic core from *MATLAB* to *C++*. The *MATLAB* AFE was written with a strong object-oriented approach, and it would seem natural to start from this robust *MATLAB* code to generate a corresponding *C++* library. Automatic code generation using the *MATLAB* Compiler SDK[1] or the *MATLAB* Coder toolbox[2] was considered, but led to two significant issues: non-standard data types were introduced, dedicated to *MATLAB*; linking to closed source shared *MATLAB* libraries was needed to produce the executable code. The automatic *C/C++* code generation from *MATLAB* was thus discarded. It has been decided to recode a specific AFE in *C++* from scratch, but keeping highly inspired by the structure of the genuine *MATLAB* implementation.

1 `https://www.mathworks.com/products/compiler.html`.
2 `https://www.mathworks.com/products/matlab-coder.html`.

5.1.1 Standard libraries and mathematical tools

openAFE takes benefit from existing external open source libraries to propose standard and maintainable structures. These include:

- the *C++ Standard Template Library (STL)*[3], implementing generic types such as vectors, pairs, maps, smart pointers,*etc.*;
- the *BOOST library*[4] for its circular buffer implementation, exploited by *rosAFE* to represent audio signal buffers;
- the *FFTW library*[5], which is actually used inside *MATLAB*, for Fast Fourier transform computation.

In addition to these open source libraries, *openAFE* contains a class named *mathTools* which assembles *C/C++* implementations of some useful *MATLAB* functions, such as convolution, linspace, conversion from frequency to ERB, etc. These are not documented here, since they are only used internally and are not meant to be used outside *openAFE*.

Note that since the genuine *MATLAB* AFE implementation and the linked libraries are released under the copyleft *GNU General Public License*, *openAFE* is also released with the same license.

5.1.2 Signal representation

Input or output signals are described through standard *C++* classes defining attributes and methods for their parameterisation, creation and destruction. At first, a general *openAFE::Signal* class is defined, highlighting common attributes and methods inherited by the more specific signal classes described below. For instance, a signal instance is represented by its name, its sampling frequency, the channel it represents (mono, left or right channel) and its size.

All signal instances share the same signal buffer description, relying on the *Boost::CircularBuffer* class provided by the *BOOST* library, see Figure 5.1. Like many other circular buffer implementations, it has the following properties:

- the capacity of the circular buffer is fixed and does not change automatically when pushing new data inside;
- even with a constant capacity, new data can be pushed as often as required into the circular buffer; if the circular buffer is full, data are overwritten.

3 `https://www.sgi.com/tech/stl/index.html`
4 `http://www.boost.org/`
5 `http://www.fftw.org/`

Using this kind of buffer makes sense when dealing with a continuous audio data flow to avoid continuous data shift in memory. This is the case in the proposed implementation, where audio data originate from *BASS*, which continuously publishes chunks to *openAFE* (or any other client). A consequence of this circular architecture is that a single chunk of audio data may not be always stored in a contiguous memory zone. However, one circular buffer may be represented by two separate contiguous memory arrays (called *array_one* and *array_two* in the *BOOST* library). A dedicated structure called *twoCTypeBlock* has been designed to hide this implementation detail from the user, so that data inside a circular buffer can be accessed by requiring either the whole buffer, or only the last appended chunk, or only last appended N frames, or the whole new frames, through the respective methods *getWholeBufferAccesor()*, *getLastChunkAccessor()*, *getLastDataAccessor()* and *getOldDataAccessor()*.

On this basis, three new dedicated signal classes are defined, both of them inheriting from the general *openAFE::Signal* class described above (see the inheritance diagram on Figure 5.2). These three signal classes directly reproduce the signal definitions in the genuine *MATLAB* AFE, namely:

- *openAFE::TimeDomainSignal*, for one-dimensional (time) signals;
- *openAFE::TimeFrequencySignal*, which corresponds to two-dimensional signals, with the first dimension related to time and the second to the frequency channel;
- *openAFE::CorrelationSignal*, for three-dimensional signals where the third dimension is a lag position.

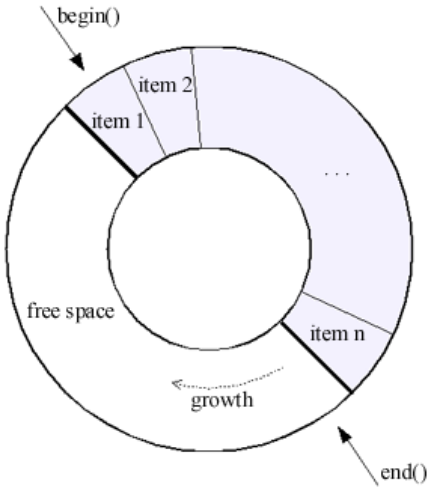


Figure 5.1: A circular buffer presentation from the *BOOST* Library.

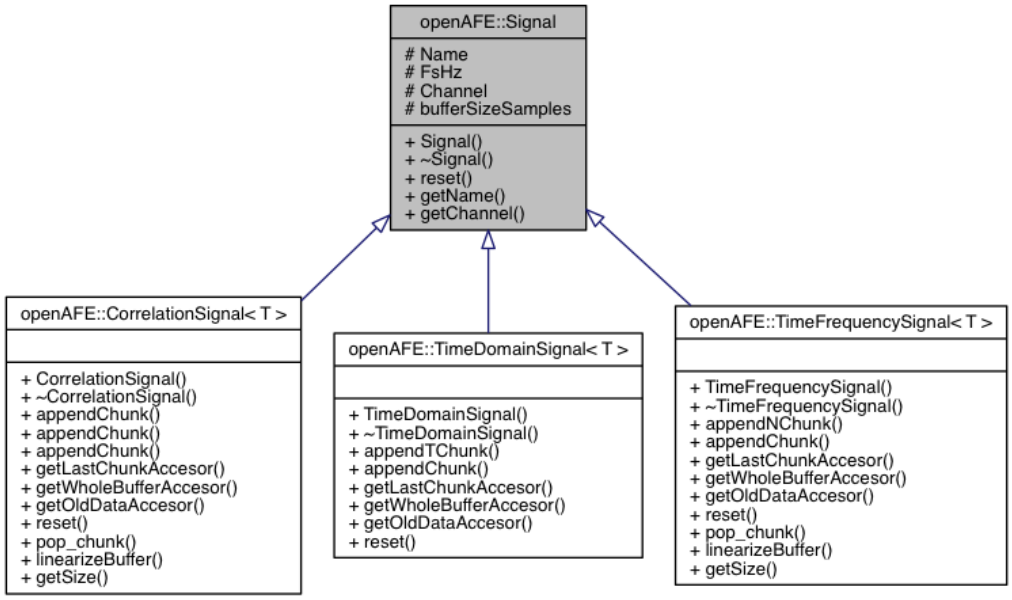


Figure 5.2: Inheritance diagram for the *openAFE::Signal* class.

5.1.3 Processor representation

The purpose of the Auditory Front-End (AFE) is to extract auditory representations from a stream of binaural audio data. In practice, each processor is responsible for one individual step in the extraction of a given representation, to be later used by higher modeling or decision stages. In the *MATLAB* AFE implementation, processors are connected to each other and form a tree: a processor whose output is routed to another processor's input is henceforth called parent while the second one is called child. Of course, a processor can have multiple children, and multiple parents as well. The *openAFE* organisation of processors is still rooted in this tree architecture, an instance of which is shown in Figure 5.3. Its current implementation is detailed below.

A Data exchange between processors

The tree workflow has been implemented through an object-oriented description, providing general methods to access data from a parent processor, process it, and release it to its child(ren). Basically, each processor instantiation contains a pointer to its parent(s), but does not include any information about its child(ren). Therefore, the data flow between two processors is handled by two general methods: *processChunk()* and *releaseChunk()*. *processChunk()* gets data from the parent processor and processes it. Once the representation is obtained, the result is stored in a private internal memory zone (attributes *leftPMZ* and *rightPMZ*), ready to be used by its child(ren). Then, *releaseChunk()* can be used to search for the last available chunk of data in this private memory zone, and to append it to the output signal of the processor. This saves children of a processor from making a local copy of their parent's output(s), as it shares a read access to its outputs(s). Avoiding

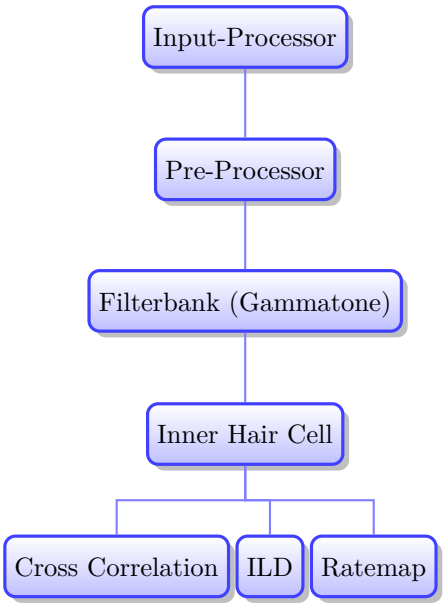


Figure 5.3: Tree of processors: each processor is represented as a box; boxes can be connected to each other. In this sample tree, *innerhaircell* is the child of *gammatone*, and *gammatone* is then the parent of *innerhaircell*. A processor can have multiple children (see the *innerhaircell* processor) and multiple parents (not illustrated here).

replication by each child(ren) limits the memory needs. Figure 5.5 shows some source code illustrating this mechanism.

B Description of processor classes

Multiple classes have been used to describe all processors. On the top of all representations is the *openAFE::Processor* class, which describes the general common properties and methods of all processors. This includes the aforementioned chunk processing methods *processChunk()* and *releaseChunk()*, but also attributes like the name of the processor, its sampling frequency, its input and output size, etc. Three classes inherit this description, respectively dedicated to time-based, frequency-based and lag-based processors. This is to put in parallel to the three signal classes *openAFE::TimeDomainSignal* (1D signals), *openAFE::TimeFrequencySignal* (2D signals) and *openAFE::CorrelationSignal* (3D signals), see Section 5.1.2. Each of them gives rise to specific processor classes, which actually implement the processor representations. For now, seven processor classes are available:

- *openAFE::InputProc* describes the very first processor of a processor tree; it receives audio from any classical C-type array and stores the data in two *openAFE::TimeDomainSignal* instances, corresponding to the left and right channels;
- *openAFE::PreProc* implements preprocessing computations, like DC-bias removal, pre-emphasis, RMS normalisation, level scaling to a pre-defined SPL reference, and

middle ear filtering; each preprocessing step can be precisely tuned by adequate parameters set through their corresponding methods;

- *openAFE::GammatoneProc* implements a bank of gammatone filters that simulates the frequency selective properties of the human cochlea;
- *openAFE::IHCProc* implements Inner Hair Cell (IHC) functionality by extracting the envelope of the output from individual gammatone filters;
- *openAFE::CrossCorrelation* implements a cross-correlation between the right and left Inner Hair-Cell signal representations, in the Fourier domain; the result is normalized by the auto-correlation sequence at lag zero, and evaluated on a given time range; it comes as a three-dimensional *openAFE::CorrelationSignal* instance, where the three dimensions respectively represent time frame, frequency channel and lag;
- *openAFE::ILDProc* implements the Interaural Level Difference estimation for individual frequency channels by comparing the frame-based energy of the left and the right-ear IHC representations;
- *openAFE::RateMap* implements the computation of a rate-map, *i.e.*, a map of auditory nerve firing rates; the rate-map is computed for individual frequency channels by smoothing the IHC signal representation with a leaky integrator.

Importantly, all the parameters of these processors, which appear as attributes of their corresponding classes, are identical to those of the genuine *MATLAB* processor implementation. Consequently, they are not listed again in this document. The reader is invited to refer to the official Two!Ears documentation. Usually a parameter is a boolean, which refers to a flag (for instance, turn on or off the DC-Removal filter for an instance of the *openAFE::PreProc* class), a value (e.g. DC-Removal cut off frequency), or a special parameter of the processor (for instance, windowing type for an instance of the *openAFE::ILDProc* class). A change in a processor parameter may require some preparation before its effective use in the processing: for instance, changing the cut-off-frequency of a filter yields to the re-initialisation of this filter. This preparation must be explicitly called after a change of parameter through the corresponding *prepareForProcessing()* method, which is described in the general *openAFE::Processor* class, and thus inherited by all processor objects. Note that each parameter is coupled with corresponding *set()* and *get()* methods, which allow to respectively set or read the parameter value of the instantiated processor. Except the blacklisted parameters listed in the original AFE, all the parameters can be changed at any time, thus allowing top-down feedback from higher levels of the Two!Ears architecture. The new parameters are immediately used for the new chunks arriving just after the modification. To conclude, the inheritance diagram for the *openAFE::Processor* class is shown in Figure 5.4. It lists, when possible, all the attributes and methods of all classes listed above.

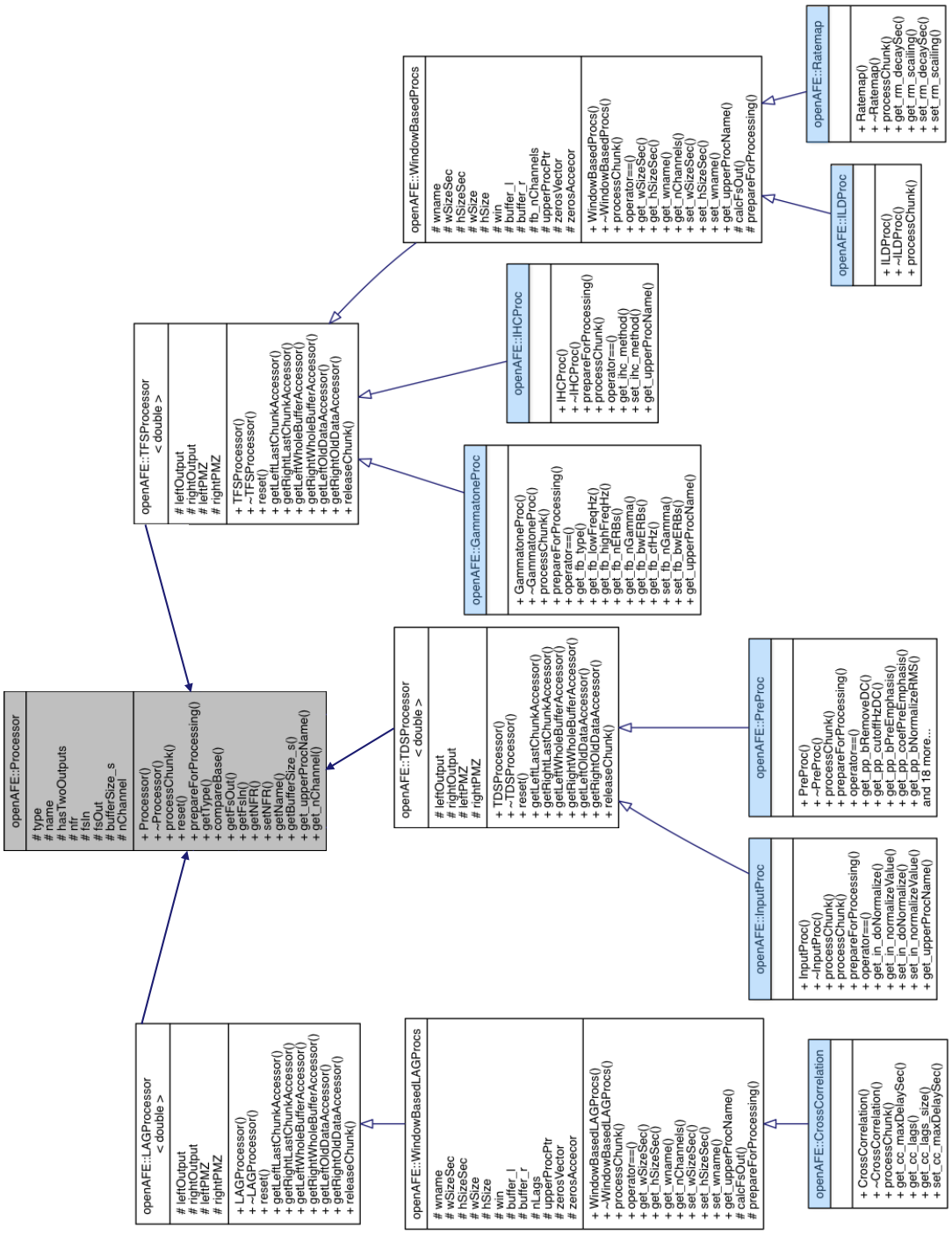


Figure 5.4: Inheritance diagram for the *openAFE::Processor* class. Classes in blue can be instantiated.

5.1.4 Some implementation considerations

A The *processorVector* class

In addition to the classes dedicated to the implementation of processor algorithms, a class *openAFE::processorVector* is introduced to handle multiple instances of a same processor. Indeed, the front-end must be able to cope with multiple realisations of the same processor, possibly tuned with different parameters. *openAFE::processorVector* is in charge of storing and managing these instantiations, by providing adequate methods to add, remove, compare and search for multiple instances of a processor. In practice, *openAFE::processorVector* stores only a list of smart pointers to each processor instantiation. As a result, and according to the proposed implementation, a processor object is destroyed as soon as it is removed from the corresponding list.

B Parallel computation

The proposed implementation highly relies on multi-threading capabilities, which is offered almost wherever possible. As soon as multiple channels are independently processed in one processor, the implementation automatically creates one thread per channel. For instance, a *gammatone* processor instantiation of 31 left and right channels creates two threads for the two binaural channels, each of them generating 31 other threads for each filter processing task. Of course, none of these threads give rise to a concurrency problem. However, some processors in charge of light computations are single-threaded, since no gain has been observed in comparison to a multi-threaded implementation. This is the case of the *ILD* and *Ratemap* processors.

C Code example

As already stated, the *openAFE* library is only in charge of providing the algorithmic core of the *ROS* implementation of the AFE. Nevertheless, it can be easily used in an independent *C/C++* linked program. One has to keep in mind that the tree of processors required to compute a given audio representation is not automatically created on request. The processors involved in the computation of the desired representation must be explicitly called successively in the source code. This is illustrated in Figure 5.5, where a cross-correlation computation is requested. Then, the *Input* processor, the *Pre-Processor*, the *Gammatone* processor, and the *Inner Hair Cell* processor must be first explicitly used to compute intermediate audio representations. Figure 5.5 also highlights the *processChunk()* and *releaseChunk()* mechanism described in Section 5.1.3-A.

```
/* Input processor instantiation */
shared_ptr <InputProc > inputP;
inputP.reset( new InputProc("input", fsHz, 10 /* bufferSize_s */,
                            false /* doNormalize */) );

/* Pre processor instantiation */
shared_ptr <PreProc > ppP;
ppP.reset( new PreProc("preProc", inputP ) ); /* default parameters */

/* Gammatone processor instantiation */
shared_ptr <GammatoneProc > gtP;
gtP.reset( new GammatoneProc("gammatoneProc", ppP ) ); /*default*/

/* IHC processor instantiation */
shared_ptr <IHCProc > ihcP;
ihcP.reset( new IHCProc("innerHairCell", gtP ) ); /*default*/

/* Crosscorrelation processor instantiation */
shared_ptr <CrossCorrelation > xcorrP;
xcorrP.reset( new CrossCorrelation("xcorrP", ihcP, wSizeSec, hSizeSec,
              maxDelaySec, wname ) );

/* Launch the successive computations of the processing tree */
inputP->processChunk ( earSignals[0].data(), earSignals[0].size(),
                       earSignals[1].data(), earSignals[1].size() );
inputP->releaseChunk();

ppP->processChunk ();
ppP->releaseChunk();

gtP->processChunk ();
gtP->releaseChunk();

ihcP->processChunk ();
ihcP->releaseChunk();

xcorrP->processChunk ();
xcorrP->releaseChunk();

/* Get the cross-correlation result in the lOut variable */
vector<vector<shared_ptr<twoCTypeBlock<double>>>> lOut =
                                  xcorrP->getLeftWholeBufferAccessor();
```

Figure 5.5: Code snippet showing the processor instantiations and parameterisation, followed by the successive methods calls required to get a cross-correlation result.

5.2 ROS implementation of the auditory front-end : *rosAFE*

The proposed AFE implementation encapsulates the *openAFE* library in a *GenoM3* component. After a formalisation of the notion of concurrency between processors, its design is outlined.

5.2.1 A short reminder on the proposed design

The algorithmic core of the AFE is made of the *C/C++ openAFE* library, which implements—when possible—multi-threading and parallel computations inside one processor. But considering the tree of processors shown in Figure 5.3, one can also highlight another level of parallelisation *between* processors. This concurrency property is discussed in the following, together with its actual implementation.

A Some considerations about concurrency between processors

From the processor tree in Figure 5.3, only the root processor, i.e. the *Input* processor, reads audio data from another component of the architecture (the *BASS* in practice). Other processors are interconnected with a parent/child structure (Section 5.1.3). This highlights two kinds of concurrency between processors.

Vertical concurrency While a processor works on a resource delivered by its parent, the parent can already prepare the next resource. This kind of concurrency concerns for instance the *Input* processor, the *Pre-processor*, the *Gammatone* processor and the *IHC* processor in Figure 5.6.

Horizontal concurrency Children of a processor are mutually independent and can process concurrently their parent's output. This kind of concurrency concerns the *Cross-Correlation* processor, the *ILD* processor and the *Ratemap* processor in Figure 5.6, all of them having the same parent (the *IHC* processor).

B Formal design and generic architecture

A processor takes an input resource from its parent and produces an output resource to its child(ren). As aforementioned, children of a same parent share a read access to a single memory zone, managed by the parent. In addition, the parent processor owns a private memory zone for its internal computation. With this memory management plan, each

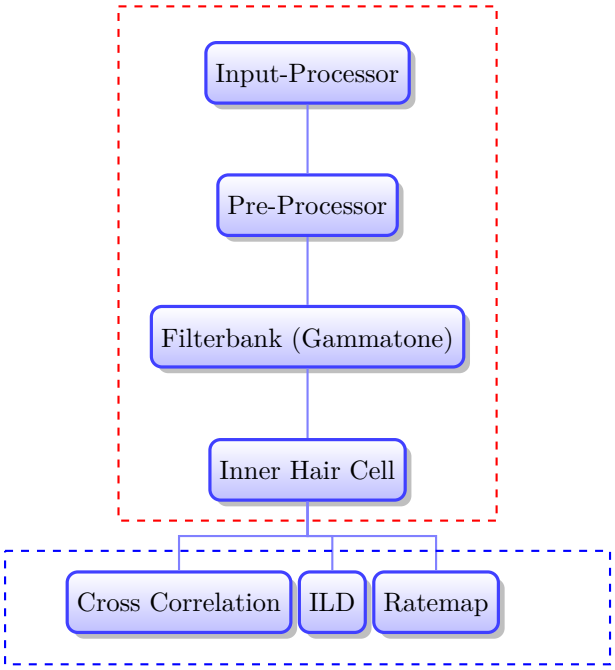


Figure 5.6: A tree of processors, leading to vertical (red) and horizontal (blue) concurrency.

processor can be formalized by a state machine similar to Figure 5.7. A processor goes through the following four distinct states, in a loop.

waitExec The processor is ready to read a new input resource, coming from its parent.

exec As soon as its parent releases the resource, the processor performs its computation. It reads the input resource from its parent's shared memory zone, and stores the result of the computation—its output resource—in its own private memory zone.

waitRelease The processor stays in a waiting state while its children are still processing the previous output resource it has released. Children lock the processor's shared memory zone.

release Once all children are done processing the previous output resource, the processor can release the new one: it copies the content of its private memory zone to its shared memory zone.

Additionally to these 4 functional states, the implementation requires the definition of the *start*, *stop* and *delete* states to respectively initialize, stop and remove a processor from the processing tree. These are not represented in Figure 5.7.

On this basis, the two aforementioned kinds of concurrency can be implemented as follows.

B-1 Vertical concurrency Considering a serial chain of three processors, the interaction between the three state machines describing them can be summarized as:

1. while in *waitExec* state, *processor 2* needs a token issued after the *release* state of its parent (*processor 1*) in order to fire the transition to the *exec* state;
2. while in *waitRelease* state, *processor 2* needs a token issued after the *exec* state of its child (*processor 3*) in order to fire the transition to the *release* state.

This is summarized in Figure 5.8(a).

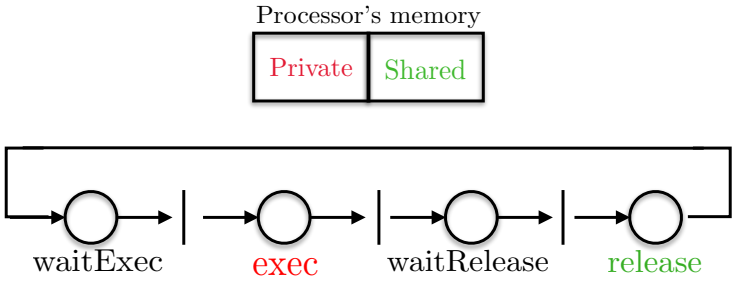


Figure 5.7: State machine and memory management of a processor.

B-2 Horizontal concurrency Considering a parallel chain of two processors, both connected to the same parent, the interaction between the three state machines describing all of them can be summarized as:

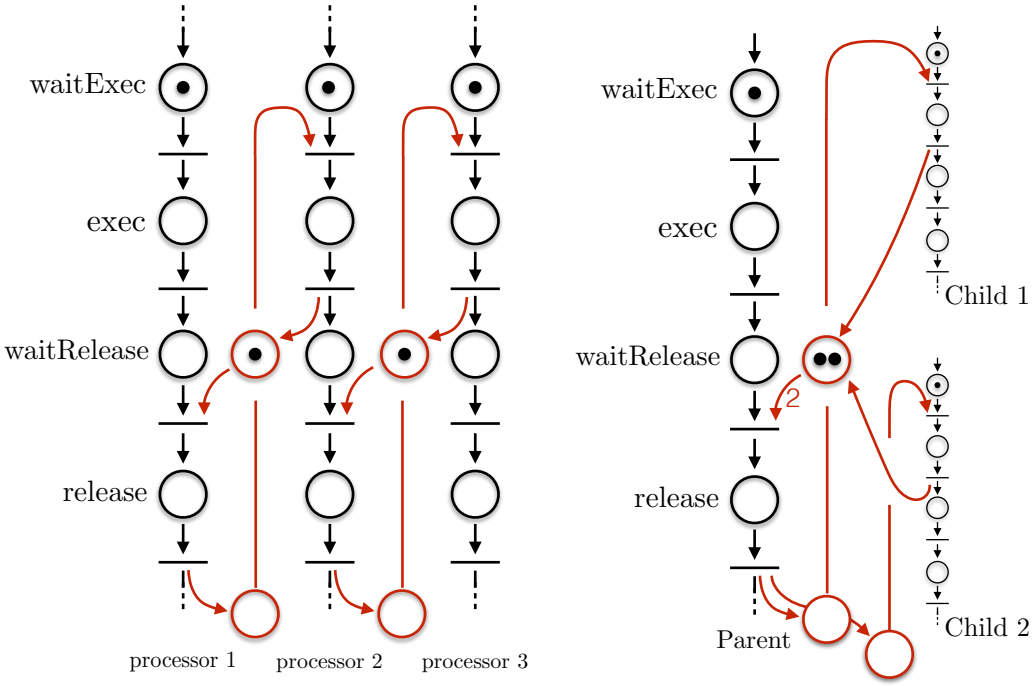
1. when the parent processor *parent* leaves its *release* state, it issues individual tokens allowing each child (*child 1* and *child 2*) to fire the transition from *waitExec* to *exec* state.

2. once a child leaves its *exec* state, it issues one token. The parent processor needs as many tokens as it has children (two, here) to fire the transition from *waitRelease* to *release* state.

This is summarized in Figure 5.8(b).

5.2.2 *rosAFE*: a ROS/GenoM module at the functional level

Contrarily to the *MATLAB* implementation of the AFE, a *GenoM3* module enables concurrent processing. In view of the many concurrency and synchronisation properties outlined in the previous section, *GenoM3* greatly eases the specification and the development of *rosAFE*. This is the subject of the next sections.



(a) Vertical concurrency **(b)** Horizontal concurrency

Figure 5.8: Petri nets for a serial (left) or parallel (right) chain of processors.

A Description of the module

The following elements are defined in the *dotgen* file of the *rosAFE* component.

Activities In view of the proposed formal design for handling processor concurrency, activities are used to actually implement the AFE processors state machine, with one activity per processor. An activity declaration inside the *.gen* file is provided in Figure 5.9. It shows that in addition to a list of specific parameters, a processor activity is also defined by the state machine depicted in Section 5.2.1-B. This implementation is generic, and every processor implementation relies on the same activity declaration.

Tasks These are in charge of executing activities. Importantly, concurrency of multiple tasks entailed in a *GenoM3* component is included in the automatically generated real time code, and is transparent to the user. For the *ROS* middleware, this is implemented as the concurrent execution of one thread per task. In the proposed implementation, each processor is associated to one activity, each of them being executed in independent tasks, *i.e.*, independent threads. Then, depending on the number of cores, the host machine can either perform parallel processing or task switching to run these threads concurrently. As indicated in the activity declaration shown in Figure 5.9, a task *preProc* is also declared in the *.gen* file, see Figure 5.10. Again, this declaration is generic, and all the processor activities have their own dedicated tasks, which are all declared as periodic, with a period of 50 ms. Note that this duration does not correspond to the time required for an activity to perform a chunk computation. If more time is needed, then the activity goes on without any problem, and is woken up again later.

Internal data structure (IDS) The IDS mainly contains a list of structures which actually implement the processor objects defined in the *openAFE* library. This allows each activity to access, through its corresponding processor instantiation, to its parameters and computed audio representations. Moreover, the IDS also contains two instantiations of a structure named *flagMap*, which contains all the synchronisation signals needed to perform vertical and horizontal concurrency. This structure is made of a list of names of the connected processors and a list of boolean flags by which the parent processors notify their children that a new resource is available, and children notify their parents that they are ready to read a new resource. The first instantiation *flagMapSt* is dedicated to the tokens required to go from the *waitRelease* to the *release* state, while the second one *newDataMapSt* allows to go from the *waitExec* to the *exec* state. A change of flag is detected through an active polling approach, where each processor periodically checks if a flag has reached an adequate value, thus enabling the transition to their next state.

Ports They are described in a separate *rosAFEInterface.gen* file, which lists all the available ports of the module, see Figure 5.12. The proposed implementation exhibits multiple out ports:

- one for each processor, which allows another component to have access to a

```
activity PreProc (
      in string name = "preProc" : "The name of this preProc",
      in string upperDepName = "input" : "The name of the upper dependencie",
      in boolean pp_bRemoveDC = FALSE : "Flag to activate DC-removal filter",
      in double pp_cutoffHzDC =  20 : "Cutoff frequency (Hz) of DC-removal high-pass filter",
      in boolean pp_bPreEmphasis = FALSE : "Flag to activate the pre-emph. high-p. filter",
      in double pp_coefPreEmphasis =  0.97 : "Coefficient for pre-emph. comp.",
      in boolean pp_bNormalizeRMS = FALSE : "Flag for activating automatic gain control",
      in double pp_intTimeSecRMS =  0.5 : "Time constant (s) for automatic gain control",
      in boolean pp_bLevelScaling = FALSE : "Flag to apply level scaling",
      in double pp_refSPLdB =  100 : "dB SPL corresponding to input signal RMS value of 1",
      in boolean pp_bMiddleEarFiltering = FALSE : "Flag to apply middle ear filtering",
      in string pp_middleEarModel =  "jepsen" : "Middle ear model (jepsen or lopezpoveda)",
      in boolean pp_bUnityComp = FALSE : "Compensation to have maximum of unity gain for
          middle ear filter (automatically true for Gammatone and false for drnl filterbanks)"
    ) {
    doc "Prior to computing any of the supported auditory representations, the input signal
        can be pre-processed with one of the following elements:
        1. Direct current (DC) bias removal
        2. Pre-emphasis
        3. Root mean square (RMS) normalisation
        4. Level scaling to a pre-defiend sound pressure level (SPL) reference
        5. Middle ear filtering";

    task preProc;
    validate existsAlready ( local in name, local in upperDepName, in ::ids );

    codel <start>       startPreProc( local in name, local in upperDepName, inout
         preProcessorsSt, inout flagMapSt, inout newDataMapSt, inout inputProcessorsSt, in infos
         , port out preProcPort, /* Arguments Of processor*/ local in pp_bRemoveDC, local in
         pp_cutoffHzDC, local in pp_bPreEmphasis, local in pp_coefPreEmphasis, local in
         pp_bNormalizeRMS, local in pp_intTimeSecRMS, local in pp_bLevelScaling, local in
         pp_refSPLdB, local in pp_bMiddleEarFiltering, local in pp_middleEarModel, local in
         pp_bUnityComp ) yield waitExec, stop;

        // The state Machine, start
    codel <waitExec> waitExec(local in name, local in upperDepName, inout newDataMapSt)
                yield pause::waitExec, exec, ether, delete;
    codel <exec> execPreProc(local in name, local in upperDepName, inout ::ids, out flagMapSt)
                yield waitRelease;
    codel <waitRelease> waitRelease(local in name, inout flagMapSt)
                yield pause::waitRelease, release, stop;
    codel <release> releasePreProc(local in name, inout ::ids, out newDataMapSt, port out
         preProcPort)
                yield pause::waitExec, stop;
        // The state Machine, end

    codel <delete> deletePreProc( local in name, inout preProcessorsSt, port out preProcPort )
                yield ether;
    codel <stop> stopPreProc( inout preProcessorsSt )
                yield ether;

    throw   e_noUpperDependencie, e_existsAlready, e_noSuchProcessor;
};
```

Figure 5.9: *.gen* extract, specifiying the *PreProc* processor as a *GenoM3* activity and highlighting the state machine structure of the activity.

```
    task preProc {
        period      50ms;
    };
```

Figure 5.10: *.gen* extract, specifiying the *PreProc* task in which the *PreProc* activity is run.

```
    ids {
        rosAFE::infos infos;

        inputProcessors inputProcessorsSt;
        preProcessors preProcessorsSt;
        gammatoneProcessors gammatoneProcessorsSt;
        ihcProcessors ihcProcessorsSt;
        ildProcessors ildProcessorsSt;
        ratemapProcessors ratemapProcessorsSt;
        crossCorrelationProcessors crossCorrelationProcessorsSt;

        flagMap flagMapSt;
        flagMap newDataMapSt;
    };
```

Figure 5.11: *.gen* extract, specifying the internal data structure (IDS) of the component.

specific audio representation; this representation is made available on a *chunk-by-chunk* basis through the *getLastChunkAccessor()* methods of the processor signal instances defined in the *openAFE* library (see Section 5.1.2); these ports are declared as being *multiple*, thus allowing multiple instances of the same processor to deliver their outputs;

- a global out port, including all the outputs from all running processors instances; this port is refreshed thanks to the *getWholeBufferAccesor()* method which applies to all processor's signal instances, also defined in the *openAFE* library.

Functions For utility purposes these are

- *getSignals()*, which allows to have access to the new audio representations computed by all running processors;
- *getDependencies()*, returning a string vector with all the processor names required to compute a given audio representation;
- *getParameters()*, which returns all the parameters of each running processor;
- *modifyParameter()*, allowing to set the value of a processor's parameter;
- *removeProcessor()*, which destroys an instance of an existing processor (to stop it);
- *Stop()*, which stops all running processors.

All these functions are of high interest for any *rosAFE* client. They are extensively used by the *MATLAB* client described in Section 5.3.

```
interface rosAFEInterface {

    port out rosAFE::dataObjSt dataObj;

    port out rosAFE::TimeDomainSignalPortStruct inputProcPort;
    port multiple out rosAFE::TimeDomainSignalPortStruct preProcPort;
    port multiple out rosAFE::TimeFrequencySignalPortStruct gammatonePort;
    port multiple out rosAFE::TimeFrequencySignalPortStruct ihcPort;
    port multiple out rosAFE::TimeFrequencySignalPortStruct ildPort;
    port multiple out rosAFE::TimeFrequencySignalPortStruct ratemapPort;
    port multiple out rosAFE::CrossCorrelationSignalPortStruct crossCorrelationPort;
};
```

Figure 5.12: *rosAFEInterfac.gen* extract, specifying the ports (inputs/outputs) of the component.

5.3 A *MATLAB* client and supervisor for *rosAFE*

Any tool able to dialog with *ROS* nodes and to connect to their ports/topics can be used as a client to *rosAFE*. But for now, only basic tasks can be envisaged: launching a processor, stopping it, etc. In other words, *rosAFE* basically encapsulates all the functionalities of the *openAFE* library inside a *ROS* node, but with all the functionalities of *ROS* and *GenoM3* concerning input/outputs specifications, tasks concurrency, etc. as a benefit. This means that the *rosAFE* client still has to dynamically configure "by hand" the processing tree, depending on the required audio representation. The main objective of this section is to describe a *MATLAB* interface of *rosAFE* which will automatically handle such considerations, by generating a tree of processors inside a single request. For instance, asking for an ILD computation must automatically instantiate all the processors required for this representation. This interface remains highly inspired by the genuine *MATLAB* AFE, and thus allows a smooth transition between the *MATLAB* and *ROS* implementations of the AFE.

The proposed *MATLAB* interface does not communicate directly with the *rosAFE* node. Instead, it exploits the *matlab-genomix* client in charge of the control of *GenoM3/ROS* components through the *genomix* and/or *rosix* servers (see Figure 3.3 page 15). This implementation is hidden to the user, so that the proposed *MATLAB* interface can be envisaged as a *MATLAB client* to *rosAFE*. It is considered as such below.

Exactly like the original *MATLAB* AFE, the proposed client entirely relies on an object-oriented framework, where two main objects are needed to extract any representation:

- a *data object*, in which the signal(s), the requested representation(s), and also the dependent representation(s) that have been computed in the process can be stored;
- a *manager object*, which takes care of creating the necessary processors as well as managing the computations.

5.3.1 Data Object

According to the AFE documentation[6], *many signal objects are instantiated by the AFE (one per representation involved and per channel). To organize and keep track of them, they are collected in a dataObject class.*

While signals are instantiated under *MATLAB* in the original AFE implementation through the *dataObject* object, the proposed *ROS* client sends request to actually instantiate them inside *rosAFE*. But in order to make these calls transparent to the user, a similar *dataObject_rosAFE* is still instantiated in *MATLAB*. It handles communications with *rosAFE*, keeps track of all *MATLAB* signals, and updates them when asked to. The

6 `http://docs.twoears.eu/en/latest/`

```
% Parameters for data object
sampleRate = 44100;
bufferSize_s_bass = 1;
bufferSize_s_rosAFE_port = 1;
bufferSize_s_rosAFE_getSignal = 1;
bufferSize_s_matlab = 10;
inputDevice = 'hw:2,0'; % Check input device ID with bass.ListDevices();
framesPerChunk = 12000; % Each chunk lasts (framesPerChunk/sampleRate) seconds.

% Data Object instanciation
dObj = dataObject_RosAFE(bass, rosAFE, inputDevice, sampleRate, framesPerChunk, ...
                bufferSize_s_bass, bufferSize_s_rosAFE_port, bufferSize_s_rosAFE_getSignal, ...
                bufferSize_s_matlab );
```

Figure 5.13: Minimal *MATLAB* code required to instantiate a *dataObject_rosAFE* object.

```
dataObject RosAFE with properties:

      bufferSize_s_rosAFE_port: 1
  bufferSize_s_rosAFE_getSignal: 1
           bufferSize_s_matlab: 10
                framesPerChunk: 12000
                    sampleRate: 44100
                        RosAFE: [1x1 genomix.component]
                          bass: [1x1 genomix.component]
                           ild: {[1x1 TimeFrequencySignal]}
                 innerhaircell: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
                    filterbank: {[1x1 TimeFrequencySignal]  [1x1 TimeFrequencySignal]}
                         input: {[1x1 TimeDomainSignal]  [1x1 TimeDomainSignal]}
                       preProc: {2x2 cell}
```

Figure 5.14: A data object instance after requesting an ILD and a pre-processor.

code needed to instantiate the data object is shown in Figure 5.13. As a result, the *dObj* variable contains seven default properties: three of them refer to different buffer sizes (in seconds), while other ones are related to the amount of frames per chunks, the sample rate, and to *rosAFE* and *BASSGenoM3* components (which are obviously connected to each other). If an ILD computation has been requested (see next subsection), then the *dObj* variable is completed with additional properties, each of them related to the signals computed by all the processors required to obtain an ILD, see Figure 5.14. These properties are automatically refreshed (*i.e.*, read from the corresponding *rosAFE* port) when accessed from *MATLAB*. This means that every computed auditory representation can be loaded to *MATLAB*, and sent to the higher, cognitive levels of the Two!Ears architecture. Those representations are appended to the signal instances ported from the original AFE as shown in Figure 5.15. These instantiations have exactly the same definition as in the original AFE, thus making transparent the use of *rosAFE* to the user.

5.3.2 The manager object

According to the AFE documentation, *the manager class is fundamental in the AFE framework. It is responsible for, from a user request, instantiating the correct processors and signal objects, and linking these signals as inputs /outputs of each processor. In a*

```
K>> dObj.preProc{1}

ans =

  TimeDomainSignal with properties:

       Label: 'time_0'
        Name: 'preProc'
  Dimensions: 'nSamples x 1'
        FsHz: 44100
     Channel: 'left'
        Data: [0x1 circVBufArrayInterface]
```

(a) An empty signal instance.

```
TimeDomainSignal with properties:

       Label: 'input_0'
        Name: 'input'
  Dimensions: 'nSamples x 1'
        FsHz: 44100
     Channel: 'left'
        Data: [441000x1 circVBufArrayInterface]
```

(b) A full signal instance.

Figure 5.15: Two *Time Domain Signal* instances: (a) no chunks are available in the object; (b) 10 seconds of signal are available.

standard session of the AFE, only a single instance of this class is created. It is with this object that the user interacts.

In the proposed *MATLAB* client implementation, the very same manager class *manager_rosAFE* as in the genuine AFE is modified to communicate with *rosAFE*. Instead of instantiating the processors directly in *MATLAB*, a request to the *GenoM3* module is sent via the *genomix* server. The related processor and signals are then automatically created and routed in *rosAFE*. In the same vein, this class implementation can handle any top/down processing requests (changes in some processors parameters, via the corresponding call to *modifyParameter()* function in the *GenoM3* module), destruction of some processors (via *removeProcessor()*) and transmission of the outputs from processors to *MATLAB* (by reading the corresponding *GenoM3* ports). In *MATLAB*, the processor is instantiated with the code shown in Figure 5.16. Differently from the *MATLAB* AFE, a processor immediately computes its output(s) when instantiated from the *MATLAB rosAFE* client. This means that as soon as data are available on its input(s), these are processed and the resulting audio representation is published on its corresponding *GenoM3* port(s), even if the proposed manager does not explicitly ask to actually process audio chunks. This was the role of the *processChunk()* method in the genuine AFE, while this method is now used to load the processed audio representation from the *GenoM3* environment to the *MATLAB* environment. From the user viewpoint, this slight change in the method is transparent, and *processChunk()* still presents the asked audio representation as output.

The proposed *manager_rosAFE* class is made of three properties, see Figure 5.17. The last two are handles to *rosAFE* and to the *dataObject MATLAB* class. The first property *Processors* is a list of all running processors, see Figure 5.18a. Importantly, all the listed

```
% Manager instanciation
mObj = manager_RosAFE(dObj);

% Add a ILD processor with default processing parameter
mObj.addProcessor('ild');
```

Figure 5.16: Minimal *MATLAB* code required to instantiate a *manager_rosAFE* object, and to launch the computation of an ILD representation in *rosAFE*.

```
manager RosAFE with properties:

  Processors: {}
      RosAFE: [1x1 genomix.component]
        dObj: [1x1 dataObject_RosAFE]
```

(a) An "empty" manager.

```
manager RosAFE with properties:

  Processors: [1x1 struct]
      RosAFE: [1x1 genomix.component]
        dObj: [1x1 dataObject_RosAFE]
```

(b) A manager with running processors.

Figure 5.17: Manager instances.

processors are actually instantiated inside *rosAFE*, and not inside *MATLAB*. This means that the object *manager_rosAFE* does not contain any processor, but only information on them. Those information are stored inside each processor description, made of all the processor parameters directly read and updated from *rosAFE*, see Figure 5.18b.

5.3.3 Code example

In order to illustrate how the proposed *MATLAB* client can be used together with *rosAFE* to compute an audio representation, the minimal code required to obtain an ILD is shown in Figure 5.19. The same figure also shows how the same auditory representation can be computed from the genuine *MATLAB* AFE. Except the data object instantiation, where some specific parameters for the *BASS* and *rosAFE* nodes must be specified, all the remaining code is identical between the two AFE implementations, enhancing a smooth transition between them.

```
K>> mObj.Processors

ans =

       input: {[1x1 struct]}
     preProc: {[1x1 struct]  [1x1 struct]}
  gammatone: {[1x1 struct]}
         ihc: {[1x1 struct]}
         ild: {[1x1 struct]}
    ratemap: {1x0 cell}
```

(a) Instance of all informations of currently running processors: 1 input, 2 preprocessors, 1 filterbank, 1 Inner Hair Cell, 1 ILD and 0 Ratemap.

```
K>> mObj.Processors.preProc{1}

ans =

                    name: 'time_0'
            pp_bRemoveDC: 0
           pp_cutoffHzDC: 20
         pp_bPreEmphasis: 0
      pp_coefPreEmphasis: 0.9700
        pp_bNormalizeRMS: 0
        pp_intTimeSecRMS: 0.5000
        pp_bLevelScaling: 0
            pp_refSPLdB: 100
  pp_bMiddleEarFiltering: 0
      pp_middleEarModel: 'jepsen'
          pp_bUnityComp: 1
```

(b) Details on the first preprocessor in the tree.

Figure 5.18: *manager_rosAFE* details: (a) list of running processors; (b) parameters read from *rosAFE* for one of the two running instances of the *preProcessor* processor.

```
fsHz = 44100;
bufferSize_s = 10;
channelNumber = 2;

% Empty Data Object
dObj = dataObject([],fsHz,bufferSize_s,
                  channelNumber);

% Empty Manager Object
mObj = manager(dObj,[]);

% IHC processor parameters (optional)
ihc_method = 'dau';
par=genParStruct('ihc_method',ihc_method);

% Adding a processor
mObj.addProcessor('ild', par);

% Request processing
mObj.processChunk();
```

```
% Parameters for data object
fsHz = 44100;
bufferSize_s_bass = 1;
bufferSize_s_rosAFE_port = 1;
bufferSize_s_rosAFE_getSignal = 1;
bufferSize_s_matlab = 10;
inputDevice = 'hw:2,0';
framesPerChunk = 12000;

% Data Object
dObj = dataObject_RosAFE(bass, rosAFE, ...
    inputDevice, fsHz, framesPerChunk, ...,
    bufferSize_s_bass, ...
    bufferSize_s_rosAFE_port, ...
    bufferSize_s_rosAFE_getSignal, ...
    bufferSize_s_matlab);

% Manager Object
mObj = manager_RosAFE(dObj);

% IHC processor parameters (optional)
ihc_method = 'dau';
par=genParStruct('ihc_method',ihc_method);

% Adding a processor
mObj.addProcessor('ild', par);

% Request processing
mObj.processChunk();
```

Figure 5.19: *MATLAB* code required to compute an ILD: (left) by using the genuine *MATLAB* AFE; (right) by using the proposed *MATLAB* client to *rosAFE* (initialisation of the two environments is not shown).

5.3.4 Evaluation and limitations of the current design

A first evaluation is proposed, related to the comparison of the outputs from each processor. It consists in computing, for the same input signal, the Root Mean Square Error (RMSE) between the *MATLAB* and *openAFE* outputs from each processor. Table 5.1 exhibits this list of RMS errors, together with the parameters set during this evaluation. It can be shown that both implementations exhibit similar outputs, with RMS errors almost close to 0 for all of them.

The results listed in Table 5.1 clearly show that similar processor outputs are computed from *openAFE*. However, one has to keep in mind that all these outputs, once used

Processor	RMSE
Input Processor	0
DC removal filter ($cutoffHzDC = 20$)	6,73e-10
Pre-emphasis filter ($coefPreEmphasis = 0.97$)	0
Binaural RMS normalisation ($intTimeSecRMS = 0.5$)	0
Level Scaling ($refSPLdB = 10$)	0
Gammatone Filterbank (default parameters)	2.5243e-12
Inner Hair Cell ($method$ = dau)	1.67e-12
Interaural Level Difference ($wname$ = hann)	9.11e-7
Ratemap ($wname$ = hann)	5.34e-15
Cross-correlation	2.3e-7

Table 5.1: Root Mean Square Errors (RMSE) on 42921 frames of audio (mean of RMS errors if the processor has more than one channel).

together with *ROS* in *rosAFE*, are published in the *ROS* environment, and must then be loaded inside *MATLAB* to be distributed to other high-level stages of the architecture. Table 5.2 lists the size (in MB) of each processor output(s) for 1 second of data. It can be seen that depending on the audio representation, transferring the output of one processor to *MATLAB* can represent a huge amount of data, which must be handled by the *matlab-genomix* bridge. But this interface between the *ROS* and *MATLAB* worlds exploit a TCP/IP connection model, which does not allow to envisage the simultaneous transmission of all processor outputs at the same time in a guaranteed time interval. But independently from these *MATLAB/ROS* communication considerations, all the processors simultaneously publish their outputs in real-time: for instance, if *rosAFE* is parameterized to request 12000 frames at once from *BASS* at a sampling frequency of 44100 Hz, all the processors actually publish their computed auditory representations at approximately 3.675 Hz (44100/12000), as expected. One solution is then to request the output from one processor at a time, which is then transmitted to *MATLAB* in real time without any issue.

A final evaluation consists in comparing the computation time needed by a processor for the *MATLAB* and *C++* versions of the AFE. This comparison only concerns the algorithmic part, *i.e.*, the processor implementation in *openAFE*. Table 5.3 exhibits the time required to compute each audio representation in seconds. Each value corresponds to the average computation time of 100 iterations of one processor, feeded by left and right chunks made of 2205 frames. The evaluation was conducted on a computer with an Intel 4-Core i5-4670S CPU @3.10GHz / 8Gb RAM. Results show that the proposed *C++* implementation is almost 50 times faster than the *MATLAB* version, thus highlighting the benefits of an AFE right at the functional level for the deployment system. The cross-correlation processor exhibits surprisingly low performances, showing that some code optimisation is still required here. This work is ongoing, and there is no doubt this processor will also benefit from the speedup introduced by the *C/C++* implementation of the AFE. Importantly, this evaluation only concerns *openAFE* implementations, which does not implement processor concurrency. One then can expect a better speedup with the *ROS* implementation.

Processor Type	Mean Size
Input	0.71MB
Pre-Proc	0.71MB
Gammatone	21.87MB
Ihc	21.87MB
Ild	24.96KB
Ratemap	49.88KB
Cross-Correlation	2.47MB

Table 5.2: Size of the processors output ports for 1 second of data.

Processor Type	Matlab	*openAFE*	Ratio
Input + normalisation	1e-4	7.3e-6	13.7
Pre-Proc (DC filter)	1.1e-3	7.8e-6	141.5
Pre-Proc (Pre emphasis)	1.2e-3	9.09e-6	132
Pre-Proc (RMS normalisation)	1.2e-3	3.05e-5	39.4
Pre-Proc (Level scaling)	1.6e-3	9.08e-6	176.2
Gammatone	5.1e-3	0.968e-3	5.3
IHC	2.8e-3	0.73e-3	3.84
ILD	4.3e-3	1.16e-3	3.71
Ratemap	6.8e-3	1.83e-3	3.71
Cross-correlation	26.2e-3	30.6e-3	0.86

Table 5.3: Matlab *vs openAFE* average computation time (in seconds) for each processor.

6 Components for sensorimotor and visual functions

Audio-motor source localisation can be obtained by combining binaural data streamed by the Binaural Audio Stream Server (*BASS*) with motor commands sent to the binaural head. These motor commands can even be generated in closed-loop in order to improve the information held in the localisation. This is the role of information-based sensorimotor feedback designed in WP4 and depicted into Section 4.2.7 ("b7") of Deliverable D4.3@m36. As no cognition/decision is involved in this design, it has given rise to a *GenoM3/ROS* component, which is the topic of Section 6.1.

Other knowledge sources and hypothesis-driven feedbacks designed in WP3 and WP4 rely on the incorporation of the visual modality. Consequently, specific *GenoM3/ROS* modules have also been designed to extract higher-level information from the raw stereoscopic video stream. These concern either people (Section 6.2) or objects (Section 6.3) perception.

6.1 Active audio-motor and information-based localization

Binaural sensing and motor commands of the *KEMAR* head can be jointly processed and/or interwoven so as to actively localize one source in the horizontal plane, along the three-stage framework described in Section 2.8 of Deliverable 4.2@m24, in (Bustamante *et al.*, 2015) and in references therein.

6.1.1 Implementation

Stage A implements the maximum likelihood estimation of the source azimuth and the information-theoretic detection of its activity from the short-term left and right spectrograms. Stage B assimilates these azimuths over time and combines them with the motor commands into a stochastic filter, leading to the posterior probability density function (pdf) (or "belief") of the head-to-source relative position. This stage enables front-back disambiguation and range recovery. The computed posterior pdf is an input to a feedback controller (Stage C, also termed "sensorimotor feedback") which determines the head motion leading to the next best (*i.e.*, most informative) source localization. More precisely, the admissible finite head translation and rotation are determined which lead, on average, to the minimum entropy of the posterior pdf at the next sampling time, when

the exploration is guided by the Woodworth-Schlosberg approximation of ITD between antipodal microphones over a spherical head. Roughly speaking, if the belief at time k is Gaussian and described by a 99%-probability confidence ellipse $\mathcal{E}_k$ with center E_k and minor axis $\underline{\Delta}_k$ (resp. major axis $\overline{\Delta}_k$), then the localization uncertainty at next time $k+1$ can be decreased by translating the sensor and rotating its fovea towards E_k in the direction of $\underline{\Delta}_k$ (Bustamante *et al.*, 2016c)(Bustamante *et al.*, 2016a)[1].

A *GenoM3/ROS* component named *binauloc* has implemented these ideas. It interacts with other components of the functional layer along Figure 6.1. *binauloc* takes as input the audio stream from the *Binaural Audio Stream Server* (*BASS*) and the motor flow from the modules in charge of the displacement of the kemar head and the locomotion of the mobile base. Stage A outputs a pseudo-likelihood of the source azimuth every 58 ms, on the basis of a 1-second sliding window of binaural signals. Stage B computes a Gaussian mixture approximation of the posterior pdf of the head-to-source relative position at approximately 6 Hz. The update stage of the underlying stochastic filter entails a Gaussian (unnormalized) mixture approximation of the azimuth pseudo-likelihood produced by Stage A. This approximation and the noise statistics of the prior dynamics have been empirically tuned so as to ensure reproducible and slightly conservative conclusions. The posterior weights, means and covariances produced by Stage B are published on a port of *binauloc*. Another component portrays this posterior pdf in a graphical manner by plotting the 99%-probability confidence ellipses of its hypotheses, with a color expressing their weights. *binauloc* also solves the constrained optimization problem leading to the information-based optimum head motion as per Stage C, then publishes the position profiles to be applied on the left and right wheels of the robot and on the *KEMAR* neck.

6.1.2 Experiments

The experiments simulated in Deliverable D4.3@m36 were conducted on *Jido* in an open-space 15m × 5m × 8m area delimited by dividing walls made of resin, with limited reverberation. The results of the audio-motor localization for several motion strategies as well as the genuine position of the source measured by a real-time motion capture system (with submillimetric accuracy) are displayed in Figure 6.3. A translation along the interaural axis reduces the uncertainty on the distance to the source but cannot disambiguate front from back. A pure rotation (not shown) would resolve front-back ambiguity but cannot recover the source range. A circular motion enables both azimuth and range recovery. The active motion drives the head in the same way as in the simulation.

The entropy of the moment-matched approximation of the state posterior pdf is reported in Figure 6.2. Though the circular motion is interesting for early instants, its benefits then vanish because the source is viewed from the interaural axis. Similar results ob-

1 The reference (Bustamante *et al.*, 2016b) proposed a preliminary solution to this problem by means of a gradient ascent method.

tained in the simulation are discussed in Deliverable D4.3@m36. The whole three-stage framework runs in 5 ms on the i7 quadcore @2.8 GHz 16 GB RAM laptop connected to *Jido*.

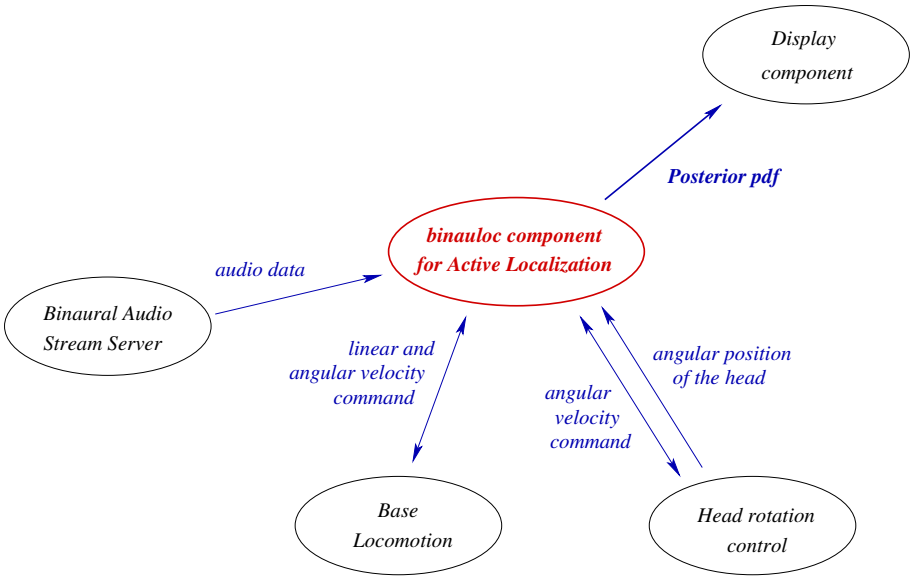


Figure 6.1: The *binauloc GenoM3/ROS* component for active binaural localization and the data flows enabling its interaction with other modules of the functional layer.

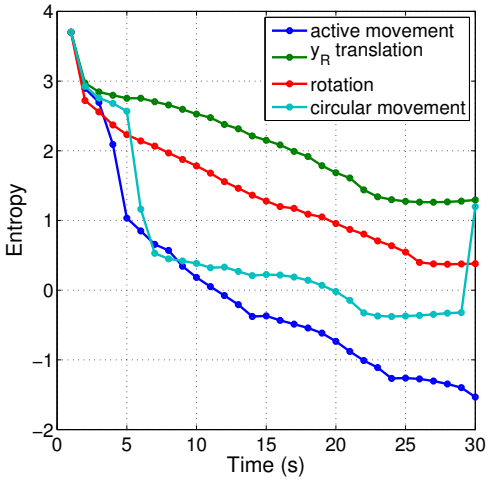


Figure 6.2: Left: Entropy reduction of the posterior state pdf for various motion strategies. Right: Screenshots of the recorded video (available at `http://bit.do/twoears-scp`) for the active motion scenario at times $t = 2\,\mathrm{s}$, $t = 10\,\mathrm{s}$ and $t = 34\,\mathrm{s}$.

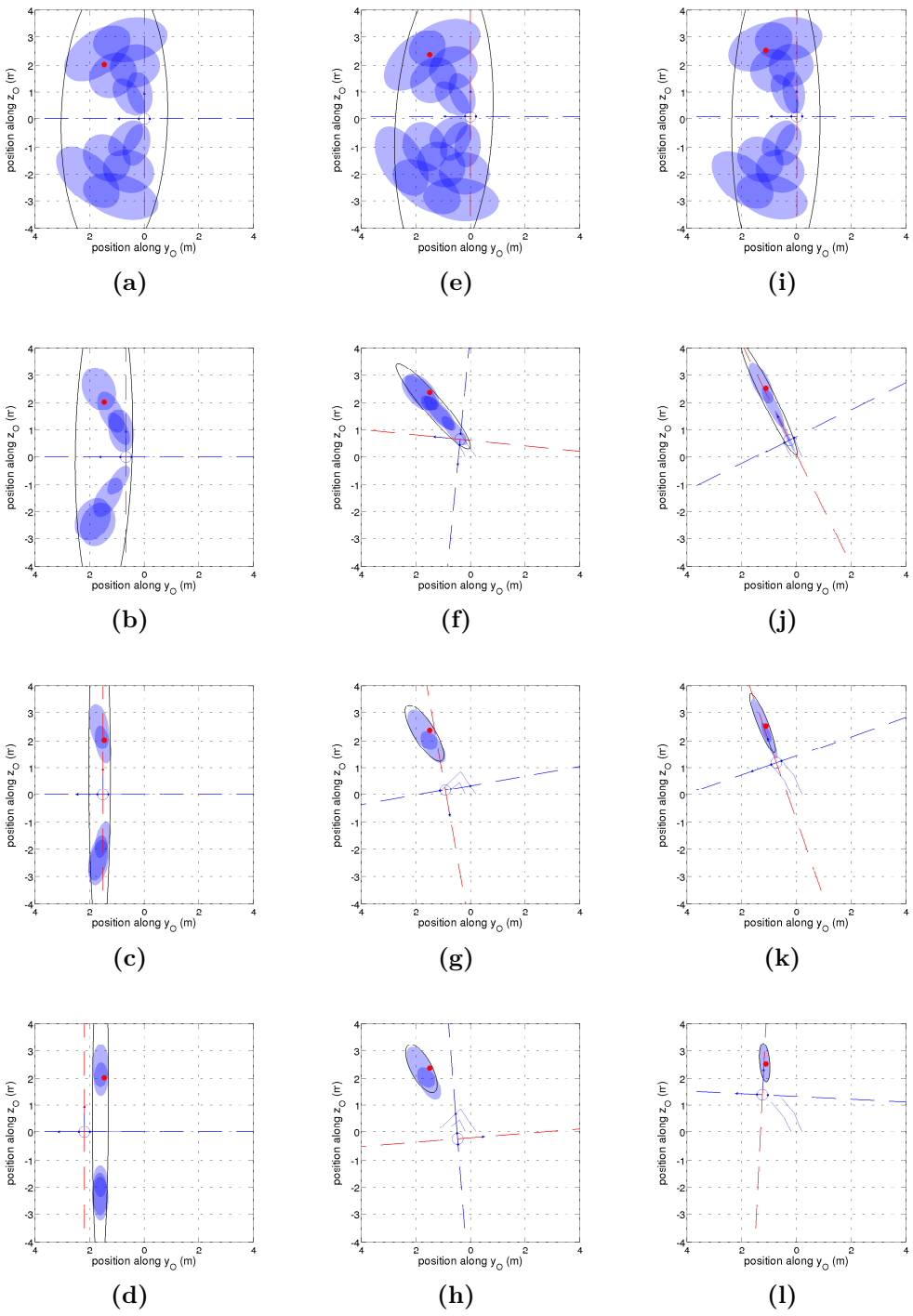


Figure 6.3: Single-source localization for different scenarios: (a,b,c,d) translation of the head along the interaural axis; (e,f,g,h) circular movement; (i,j,k,l) active motion. Snapshots (a–l) of the localization process display in the initial frame $\mathcal{F}_0$ the binaural head (front direction in dashed red, interaural axis in dashed blue): the source (in red); and the 99%-probability confidence ellipses of the hypotheses constituting the Gaussian mixture belief. The corresponding times are: (a,e,i) $t = 1\,\mathrm{s}$; (b,f,j) $t = 10\,\mathrm{s}$; (c,g,k) $t = 20\,\mathrm{s}$; (d,h,l) $t = 28\,\mathrm{s}$.

6.2 Visual functions for human detection and tracking

One *GenoM3* module was coded to provide appearance-based detection of humans in a pair of rectified stereo images, and recovery of their 3D positions. It incorporates functions from *OpenCV*'s off-the-shelf *"Haar feature based cascade classifier for object detection"*[2], which is based on a boosted cascade of classifiers (Viola and Jones, 2001). On this basis, another module was implemented that also incorporates tracking.

The steps followed in the *GenoM3* modules are the following:

1. Perform detections on left and right images;
2. Match detections between the left and right images in two steps:
   2.1. Perform template matching;
   2.2. Check if the consequent pairs of detections are on the same epipolar lines;
3. (If tracking is required), track detections along time;
4. By triangulation, recover the positions of the 3D points which led to matched detections.

6.2.1 Detection

Even though the Viola-Jones algorithm was initially designed for face detection, any object can be detected by training a cascade of classifiers. The key to get an efficient cascade is to train it on a suitable dataset. *OpenCV* includes several ready-to-use cascades (Table 6.1) as they have been trained on very good datasets, making them robust with very high detection (true-positive) rate and very low false-positive rate.

Frontal face	Profile face	Eyes
Mouth	Nose	Ears
Upper Body	Lower body	Full body

Table 6.1: Robust trained cascades included in *OpenCV*.

The *GenoM3* modules rely on an `.xml` file listing the potentially used cascades. By default, the first element of the list is selected, but this behaviour can be changed online. This list may include cascades shown in Table 6.1 as well as custom user-trained cascades. Note that only detection is performed (*e.g.* "faces" *vs* "non-faces"), but not recognition.

2 `http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html`

6.2.2 Matching

Detections from the left and right images have to be accurately matched in order to triangulate a meaningful 3D point. This is done in two steps. First, each pair of left and right detections are compared with *OpenCV's Template Matching*[3]. The underlying similarity measure is the *Normalized Cross-Correlation* (Lewis, 1995). Whenever its value (in the interval $[0; 1]$) gets greater than a user-selected threshold, a match is detected and the second step starts. It consists in checking whether the matched detections lie on the same epipolar line, up to a user-defined tolerance (in pixels) related to the desired accuracy. So, a convenient calibration of the stereo rig must have been done beforehand.

6.2.3 Tracking

Tracking is performed by means of the *Decentralized Particle Filter* (DPF), a multiple-target tracker based on the "tracking-by-detection" paradigm, entailing one tracker per target. Its is based on the well-known *ICondensation* (Isard and Blake, 1998) particle filter. This sequential Monte Carlo approach improves the efficiency of the weighted particle approximation of the posterior pdf by means of an importance sampling function taking into account detections. A variation of the DPF implementation presented in (Moussy *et al.*, 2015) is employed. It entails a rich multi-template appearance model, which leads to higher true-positive and lower false-positive tracking rates.

Tracking is only implemented on the left camera, but only considering these detections which also matched in the right camera through the process described above. This saves computation time without limiting performances. The unique ID assigned to each track on the left video sequence is then reported on the right stream.

This DPF tracker code is not published under an open-source licence.

6.2.4 Triangulation and Publication of 3D information

Equations from Section 4.3.2 enable the recovery of the X-Y-Z positions and azimuths of the 3D points which have given rise to matched pairs of left and right detections. The (U_i, V_i) image coordinates are just set to the centers of the bounding boxes drawn in the detection process. The recovered 3D positions are published on a *ROS* topic into a buffer whose size changes along time, depending on the number of detected people and the size of the history to be published. In other words, a list of structures similar to the following is published at each time:

3 `http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html`

- Frame number
- Timestamp
- Detections
  - ID
  - 3D coordinates.

6.3 Visual functions for object detection and localization

Deliverable D5.2@m24 presented the development of *ROS* packages for the learning and detection of objects on the basis of dense 3D point clouds coming from a depth sensor. Though the underlying *Linemod* multi-modal detection algorithm led to good results, its performance significantly dropped when using the stereoscopic sensor suited to the *KEMAR* head, due the the sparsity of the reconstructed 3D point cloud. So, appearance based methods for object detection and segmentation have been looked for. To this aim, a *GenoM3* module was coded. It reads the topic published by the off-the-shelf *find_object_2d*[4] *ROS* package (Labbé, 2011) suited to multiple-object detection, then performs triangulation, then publishes the results in the same way as it was done for human detection in Section 6.2.

6.3.1 Detection

The *find_object_2d ROS* package incorporates *OpenCV*'s implementations of several vision based feature detectors and descriptors, such as SIFT, SURF, FAST and BRIEF. These detectors and descriptors, as well as many other parameters, can be chosen by the user through a very intuitive GUI, as shown in Figure 6.4. After extensive trials, the combination of *Features from Accelerated Segment Test* (FAST) detector (Rosten and Drummond, 2006) and *Binary Robust Independent Elementary Features* (BRIEF) descriptor (Calonder *et al.*, 2010) was chosen as it gave the best compromise in terms of speed and robustness to perception conditions.

find_object_2d relies on an object's database. The user can build it from images taken beforehand or online (Figure 6.4). The building process consists in placing the object in front of the camera and taking a picture of it within the GUI. Then, the region to detect is selected and added to the database. At this stage, it is possible to know the number of features which can be detected from the image. The more features there are, the better the detection will be.

Figure 6.5 shows how detections work. Each object from the database is assigned a unique ID number and a bounding box around it is drawn when it is detected. The IDs as well as

4 `http://wiki.ros.org/find_object_2d`

the coordinates of the bounding boxes of the detected objects are published on a single *ROS* topic.

For the purpose of Two!Ears, which involves a moving robot, multiple images of each object taken from different angles and distances showed the best results. This approach leads the GUI to consider multiple objects while this is in fact the same one. This is solved in the *GenoM3* module. Once all the images related to an object have been taken, the user writes a file associated to this object, containing the associated IDs, and names it with a common label. For example, if the database is composed by five images of a phone (IDs 0 to 4) and five images of a loudspeaker (IDs 5 to 9), then the file "phone" contains the numbers 0 to 4 while another file "loudspeaker" contains the numbers 5 to 9. So, the *GenoM3* module extracts all the detected IDs on the topic published by *find_object_2d*, and has just to extract the label(s) they correspond to.

As mentioned before, multiple images of the same object imply multiple detections of this object. But this easily solved by considering that the object lies within the area where all the bounding boxes overlap.

6.3.2 Triangulation

Triangulation is dealt in the same way as described in Section 4.3.2. 3D positions are retrieved on the basis of the centers of bounding boxes, then published on a *ROS* topic. The resulting 3D location is not as accurate as the results depicted for the triangulation of 3D points because the center of the bounding box in the two images does not really correspond to the same 3D point in space.

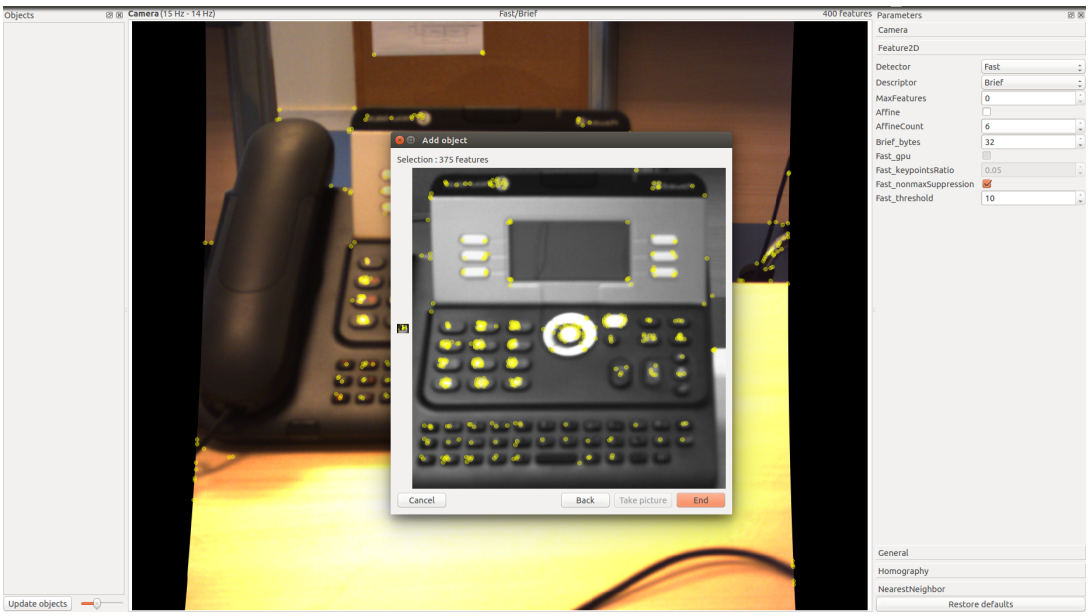


Figure 6.4: *find_object_2d*'s GUI. Adding an object to the database.

Figure 6.5: How detections are shown on the *find_object_2d*'s GUI.

7 Appendix

7.1 *Odi* documentation and user manual overview

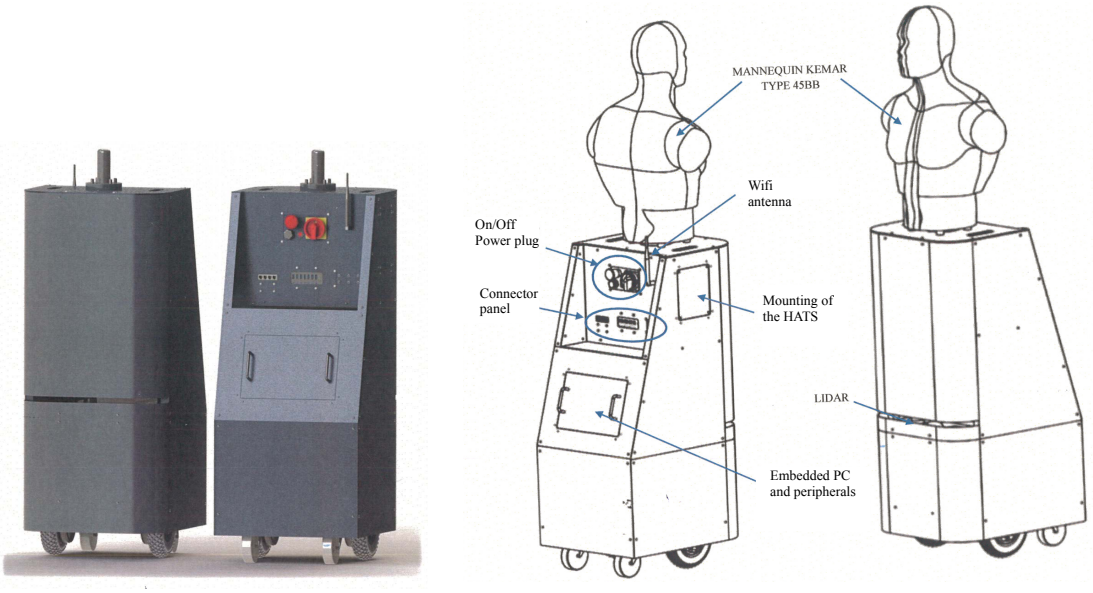
This appendix deals with the *Odi* platform, and provides all the details needed to make it ready for experiments with the Two!Ears system.

7.1.1 *Odi* preparation

A Unpacking

Odi is made of the following parts:

- 1 mobile platform,
- 1 battery charger,
- 8 JACK cables 5.5x2.1mm,



(a) *Odi* design **(b)** *Odi* overview

Figure 7.1: Details on the *Odi* platform.

- 1 KEMAR HATS support,

everything being packed inside a single package, see Figure 7.2. To unpack the overall system:

- put the package on the floor (use the "UP" indicators on the package to put it on the correct side),
- use an electric screwdriver to remove all the screws on the front and back sides of the package,
- remove all the straps on the mobile platform,
- pick all the boxes containing the HATS supports, battery charger, etc. out of the package,
- straighten the platform (you will need at least 2 persons, the robot weighting about 50kg).

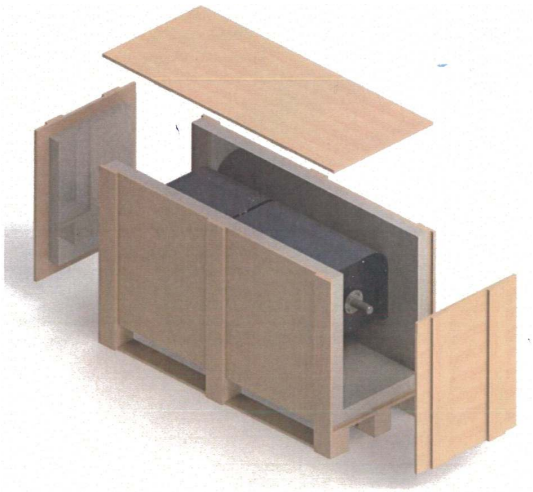
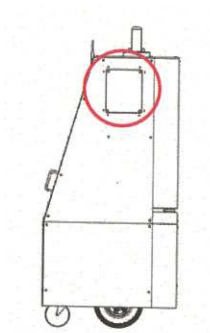
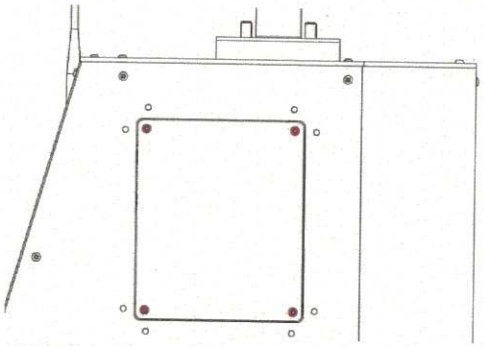


Figure 7.2: *Odi* package.

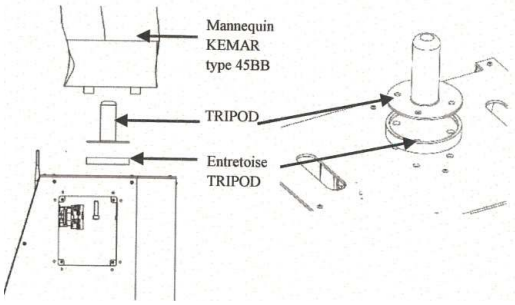
B Mounting the KEAMAR HATS on *Odi*



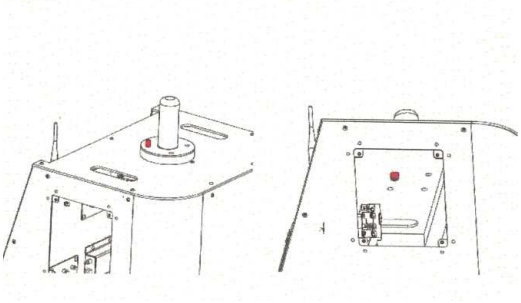
Step 1: The access to the HATS support is on the right side of the robot.



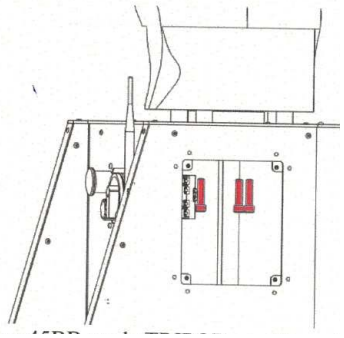
Step 2: Unscrew the hatch with the adequate screwdriver.



Step 3: Place the provided brace and the tripod in the indicated order above. Be careful to correctly align the 4 holes.



Step 4: Insert one screw inside one of the hole of the brace and tripod and check the concentricity of the system.

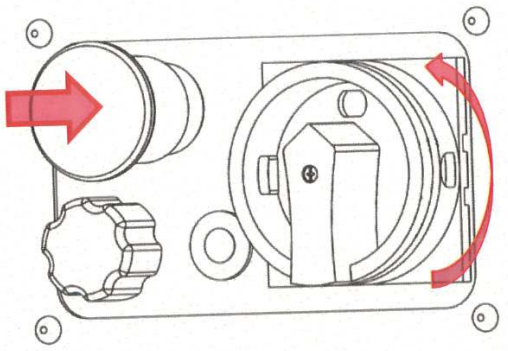


Step 5: Place the KEMAR HATS on the tripod by aligning the holes, then insert the screws and fix them. Finally, close the hatch.

C Access to the embedded computer, and peripheral devices installation

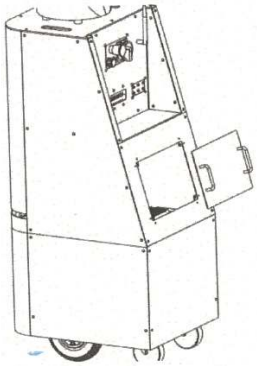
Step 1:

- The ON/OFF switch at the back must be in the STOP position (vertical position).
- The emergency switch must be ON.



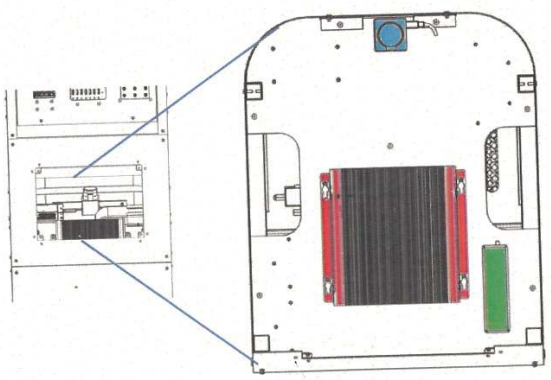
Step 2:

- Open carefully the hatch on the back of the robot. The hatch is maintained closed thanks to magnets.



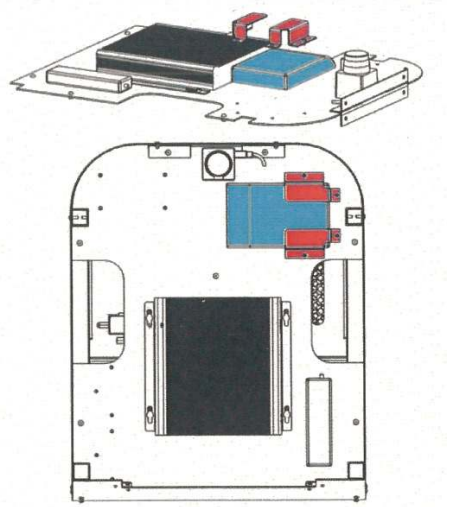
Step 3:

- The rack for peripheral devices is just in front. It initially embeds the computer (in red), the wifi module (in green) and the LIDAR interface (in blue).
- This rack contains also space for the audio Babyface USB interface, the 2 microphones conditioner systems, and the head motor controller.



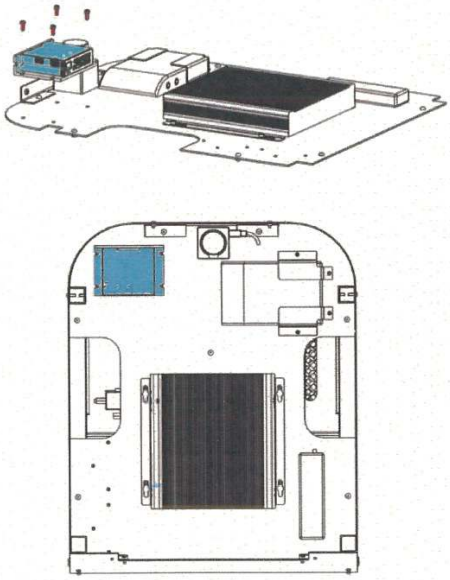
Step 4: babyface installation

- Put the RME Babyface USB audio interface where indicated and use the supplied connectors to fix it on the rack.
- The USB cable coming out the Babyface must be directly plugged in the embedded computer on one of the available USB port.



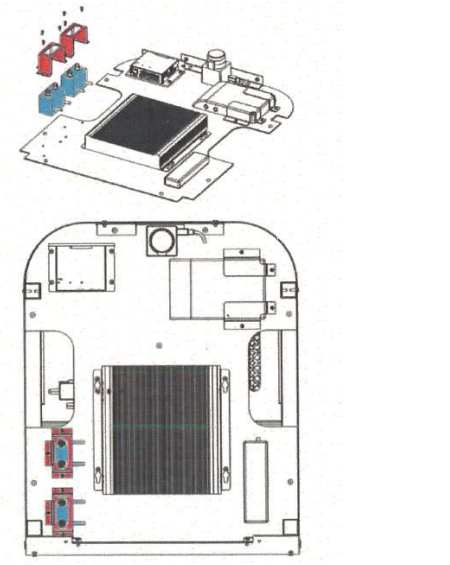
Step 5: head motor controller installation

- Put the motor controller where indicated and use the supplied connectors to fix it on the rack.
- Have a look on §D to connect the power supply to the motor controller.
- The 24V power line must be connected to the motor controller.



Step 6: microphone conditioners installation

- Put the two microphone conditioners (MMF M28) where indicated, and use the supplied connectors to fix them on the rack.



D Internal power supplies

Opening the hatch at the back of the robot allows to have access to all power supplies lines, providing 5V, 12V and 24V (regulated). Be careful to first turn off the overall system thanks to the switch at the back of the robot before accessing the power supplies.

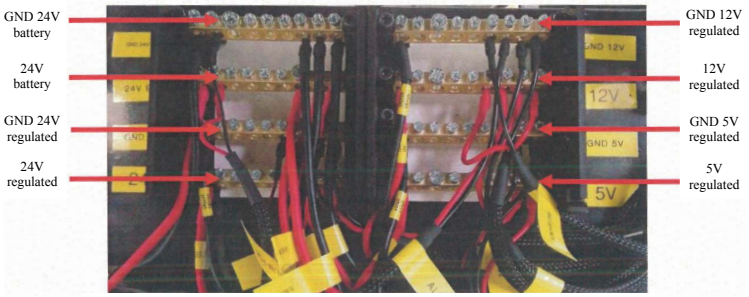


Figure 7.3: *Odi* power supplies, exhibiting the 5V, 12V and 24V voltage lines.

E *Odi* rear panel

The rear panel is detailed in Figure 7.4. Note that a push on the emergency stop button automatically stops the motorization of the robot, but does not power off other devices.

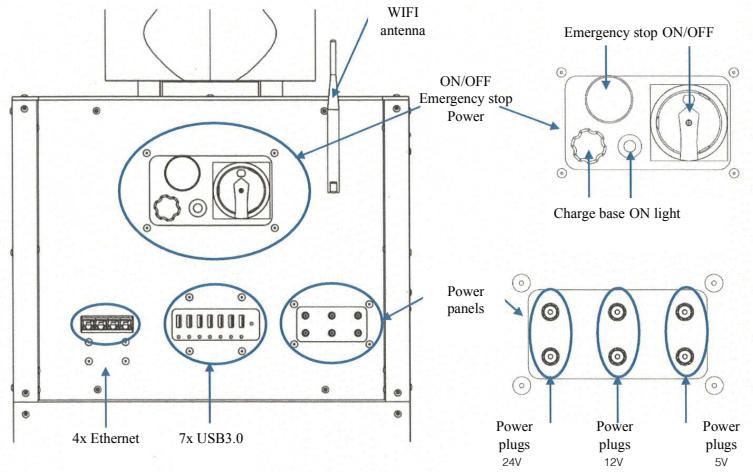


Figure 7.4: *Odi* rear panel, showing all the connectors available from outside the robot.

F *Odi* maintenance

The battery inside the robot is a lead-acid battery with a capacity of 408Wh. It enables the robot to operate autonomously for about 6 to 8 hours, depending on the robot task and the peripheral devices used. In order to maximize the battery lifetime, the robot must be charged directly after each use (please avoid deep cycles which can significantly degrade the battery lifetime).

To recharge *Odi* battery :

- verify that the robot is turned off,
- use the provided battery charger and connect it to a wall socket or any other electrical outlet,
- connect the charger outlet inside the adequate connector on the rear panel (see Figure 7.4),
- unplug the charger once the charging light turns green.

A full charge takes about 2 to 5 hours, depending on the battery state.

7.1.2 *Odi* software aspects

A Peripheral devices software installation

A-1 RME Babyface This USB audio device works out of the box with the legacy driver on Linux. This is made possible by updating the Babyface firmware to any version up to v200. If the audio device is not automatically recognized by the system, one then must put it in legacy mode manually. This is done by pressing simultaneously the Select and Recall buttons while reconnecting the USB device to the computer. Then, the "class compliant" mode is started on the device, allowing it to be recognized as a standard audio interface by the operating system.

On *Odi*, the RME Babyface is automatically recognized as the 2nd USB hardware device, which can then be directly accessed to by using the Audacity software[1] or the `arecord` command line utility. Note that to have access to the device, the user must type in any console:

```
> sudo usermod -a -G audio user
```

A-2 Webcam *Odi* is not provided with any vision system. Nevertheless, almost any webcam can be easily used to access visual information. Like for the audio device, the user must be authorized to access the device first:

1 `http://www.audacityteam.org/`

```
> sudo usermod -a -G video user
```

Then, if the pictures to be captured must be used with *ROS*, one has to install the `usb_cam` and `usb_cam_node` on the system:

```
> sudo apt-get install ros-indigo-usb_cam ros-indigo-image-view
```

B How to use *Odi*

B-1 Connection to the robot *Odi* emits a wifi network with the SSID `kemar_wifi`. Once connected to it, the IP parameters must be specified manually (no DHCP server is installed on *Odi*):

- IP adress: 192.168.11.XX (XX=between 2 and 255, with 100 excluded),
- Netmask: 255.255.255.0,
- Default gateway: 192.168.11.1

ROS also requires the following environmental variables to be set:

- ROS_HOSTNAME=192.168.1.XX (XX= the value set above),
- ROS_MASTER_URI=http://192.168.11.100:11311,
- ROS_IP=192.168.1.XX (XX= the value set before).

On the robot, modify the file `/etc/hosts` by adding suitable IP address and hostname. On the local computer, the same can be done, by adding the line

```
192.168.1.100 kemar_base
```

in the local file `/etc/hosts`.

B-2 Install the *OdiROS* packages on the client *Odi* is shipped with a CD containing an archive file `src-kemar.tar.gz`. Once *ROS* is installed on the local machine (see the Two!Ears documentation online to have more information on how to do that), extract the archive inside the workspace/src *ROS* environment (usually the `~/catkin_ws` folder). Then, install the following *ROS* packages:

```
> sudo apt-get install ros-indigo-roboteq-* ros-indigo-joy
```

and execute `catkin_make` inside the workspace/src *ROS* environment folder. Note that the following line must be commented in the file `kemar_base.launch` before actually using it (see below):

```
<include file="$(find kemar_robot)/launch/invert_odom.launch"/>
```

B-3 Move the robot First, the user must be connected to the robot via SSH. Once logged in, launch `roscore`, the *OdiROS* node `kemar_robot` and execute the startup script `kemar_drive.launch`:

```
> roslaunch kemar_robot kemar_drive.launch
```

The robot can now be controlled from the keyboard on the client by launching:

```
> roslaunch kemar_teleop kemar_keyboard.launch
```

An xbox controller can also be used to move the robot. The user then must first setup the controller:

```
> sudo apt-get install ros-indigo-joy
```

Then, the xbox controller must be connected through USB, and the command `ls /dev/input` must be invoked to find the controller device ID (look for `jsX`, where `X` is the controller ID number). To access the device, type:

```
> sudo chmod a+rw /dev/input/jsX
```

To test if the controller is correctly recognized by the system, use the `jstest` tool which should display each pressed button:

```
> sudo jtest /dev/input/jsX
```

One can now launch the *ROS* node interface with the xbox controller:

```
> rosparam set joy_node/dev "/dev/input/jsX"
> rosrun joy joy_node
```

The key pressed captured by *ROS* can be seen with:

```
> rostopic echo joy
```

B-4 How to create a navigation map On the client, install the following *ROS* package:

```
> sudo apt-get install ros-indigo-map-server
```

Then, on the robot, turn on the motors and the LIDAR:

```
> roslaunch kemar_robot kemar_driver.launch
> roslaunch kemar_robot kemar_base.launch
```

On the client, launch the following command:

```
> roslaunch kemar_navigation gmapping.launch
```

The robot can now be moved freely in the environment, by using the keyboard or the xbox controller. The map automatically created by the system can be visualized on the client by using:

```
> rosrun rviz rviz
```

The GUI can be used to select the topic `/map`. Once the entire environment is explored, the map can be saved with

```
> rosrun map_server map_saver -f filename
```

B-5 Navigation On the robot, the filename of the map obtained previously must be indicated by editing the content of the file `move_base_digitalarti.launch`. It can also be passed as an argument to each command. Then, the motors, LIDAR and the map computations can be turned on with:

```
> roslaunch kemar_robot kemar_drive.launch
> roslaunch kemar_robot kemar_base.launch
> roslaunch navigation_seb move_base_digitalarti.launch
> roslaunch navigation_seb server_localisation.launch
```

On the client, launch:

```
> roslaunch navigation_seb view_navigation.launch
```

A GUI opens (see Figure 7.5), on which the robot position can be initialized by clicking on the “2D pose estimation” button, and then by clicking on the approximate position of the robot in the map. Proceed by clicking on the "2D Nav Goal" button, and by clicking on the position to be reached by the robot.

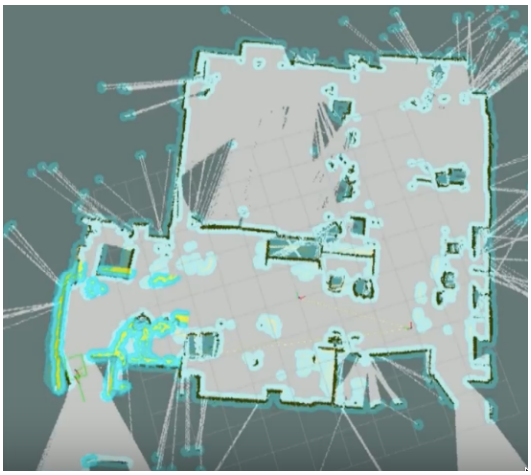


Figure 7.5: GUI interface showing the map and the robot navigating inside.

7.2 Assembly instructions for a motorized *KEMAR* HATS

7.2.1 Mechanical assembly

By default, the *KEMAR* Head-And-Torso Simulator has a movable neck that requires manual operations. For the Two!Ears project we have proposed to motorize the neck. The mechanical link between head and torso has been modified, preserving neck thickness and shape. The held solution is composed by three aluminium pieces (Figure 7.6). The mechanical design is available as open-source files. The steps to complete in order to assemble the mechanical components of the new neck are the following.

1. Remove the original part below the head and place the new one (blue colored on Figure 7.6). Keep and reuse the same screws. Once fixed, an anti-reflection adhesive film must be stuck to prevent any reflection from the proximity optical sensors below. The two audio cables can be passed through the arc-shaped hole (Figure 7.7).

2. Remove the original black piece fixed on top of the torso (the one with angular marks). Keep the screws.

3. Assemble the motor and the new aluminium part (red colored in Figure 7.6) with four M4 screws. Consider the arc-shaped hole as the rear of the new neck.

4. Place the two proximity optical sensors (*VTF180* from *Sick*) into the dedicated holes (one on front, the second on left side). The nuts provided with the sensors are too big to be placed direcly. Nuts must be rounded first to be integrated.

5. Place two plastic screws for head alignment, then position the last element on the motor shaft. There is a groove on the shaft. Align the two hexagonal screws with the groove and fix them (green colored on Figure 7.6).

6. Before placing the motor block set at the top of the torso, optical sensor sensitivity must be adjusted on both sensors. Place the head and supply the sensors with +24V DC. Check that output sensors toggle when the head turns, otherwise tune the yellow screw on sensors to adjust. Then remove the head.

7. Place the motor block set at the top of torso and screw it (Figure 7.8).

8. Pass the audio cables into the torso, place the head on the neck and screw it.

7.2.2 Electonics assembly

In order to control the *KEMAR* head some components are necessary:

- 100W brushless DC motor (*SA01ACN-8*[2]) with gearhead (*PGE12/1 i=9*);

2 `http://www.a2v.fr/program/sa01acn.htm`

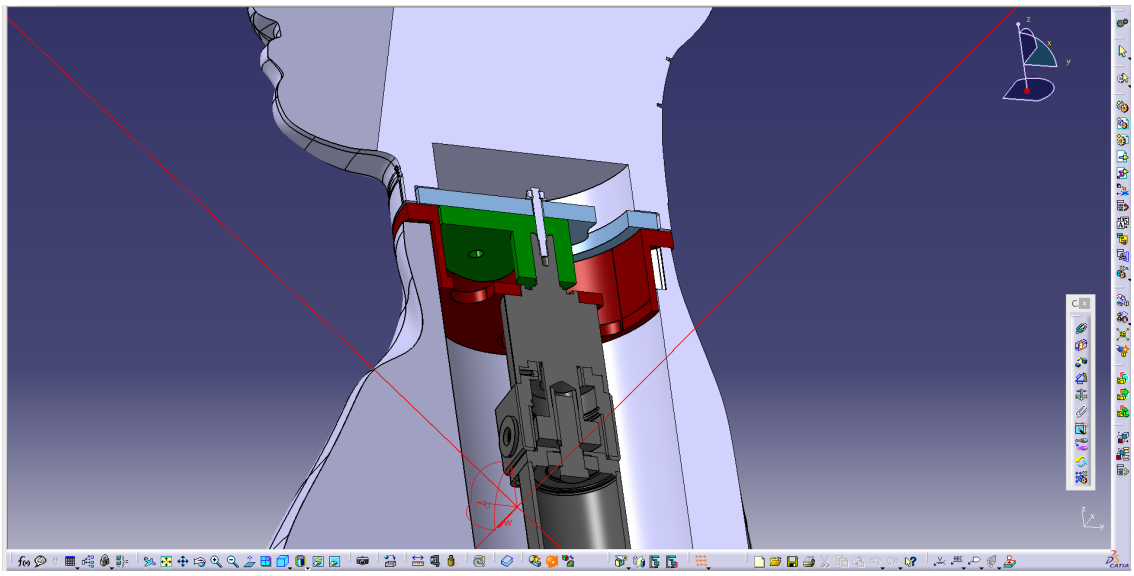


Figure 7.6: *KEMAR* side cut. From top to bottom: Blue part is screwed below the head. Green part is inserted on the motor shaft. Red part is screwed on the torso. Motor and gearhead (in black) are vertically screwed with four M4 screws.

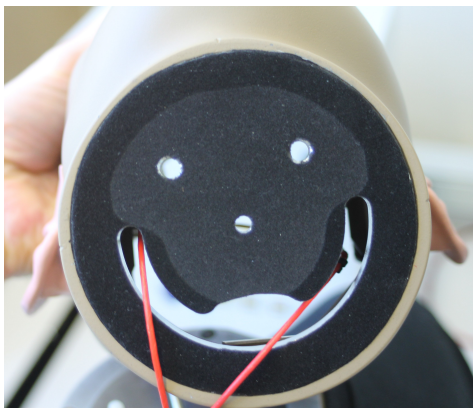


Figure 7.7: *KEMAR*'s head back sight. New mechanical part is fixed and anti-reflection adhesive film is stuck. The red audio cables are also visible.

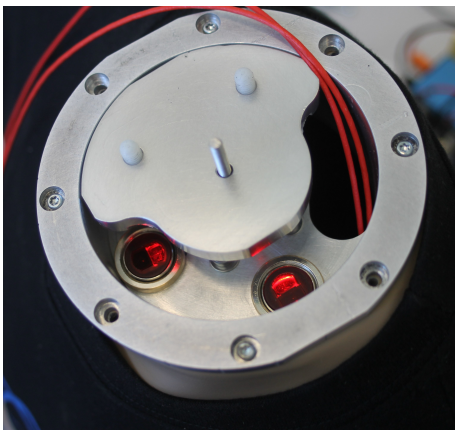


Figure 7.8: *KEMAR*'s torso with new mechanical devices. The two orthogonal sensors (red lights) are visible in the foreground. In the background the red audio cables coming from head.

- Servo controller (*ELMO Harmonica HAR5/60C*[3]);
- Encoder cable for Harmonica (*CBLIC-BAS-J3*);
- Power distribution cable (*IA1507/2M*);
- Auxiliary +24V DC cable for Harmonica (*CBLIC-BAS-J4*);
- Two proximity optical sensors (*VTF180-2N41112 from Sick*[4]).

A Assembly of the limit detector PCB

The PCB is a 2 layers stack made from standard FR4 substrate. The aim of this board is to connect the proximity optical sensors and compute simple logic to detect left & right limits (see equations7.1 and 7.2):

$$Right\ Limit\ Detection = \overline{Front\ Sensor\ +\ Side\ Sensor} \tag{7.1}$$

$$Left\ Limit\ Detection = \overline{Front\ Sensor\ +\ \overline{Side\ Sensor}} \tag{7.2}$$

The components listed in Table 7.1 are necessary to assemble the PCB. Components references are identified on PCB silkscreen. Gerber files are provided as open-source files.

B Connecting the *ELMO Harmonica* Controller

Before switching the power supply ON, be sure to have connected all the listed elements.

1. Connect the *ELMO Controller* (*J3* connector) and the motor block set (incremental encoder + Hall effect sensors) using cable *CBLIC-BAS-J3*.
2. Connect limit detection PCB (*J6* connector) to *ELMO Controller* (*J5* connector).
   - J6.1 $\longleftrightarrow$ +24V DC power supply,
   - J6.2 $\longleftrightarrow$ Power supply return,
   - J6.3 $\longleftrightarrow$ J5.6 (*ELMO* Programmable input 6),
   - J6.4 $\longleftrightarrow$ J5.8 (*ELMO* Programmable input return),
   - J6.5 $\longleftrightarrow$ J5.5 (*ELMO* Programmable input 5),

3 http://www.elmomc.com/products/harmonica-main.htm
4 https://www.sick.com/us/en/photoelectric-sensors/photoelectric-sensors/v180-2/vtf180-2n41112/p/p226906

Item	Qty	Ref	Designator	RS Ref	Farnell Ref	Mouser Ref
1	1	C1	Capacitor, X7R, 100nF, 50V	852-3273	1469310	594-K104K15X7RF53L2
2	3	D1, D2, D3	Zener Diode, 15V, 1/2W- *1N5245*	805-0189	1612374	512-1N5245BTR
3	1	J3	Pluggable Terminal Blocks, 8Pos, 3.81mm pitch, Through Hole Header	220-4888	3913120	651-1803484
4	1	J6	Pluggable Terminal Blocks, 6Pos, 3.81mm, pitch Through Hole Header	220-4872	3913119	651-1803468
5	2	J4, J5	3Pos Vertical Pin Header	745-7068		855-M20-9990346
6	2	M2, M3	MOSFET N-Channel, 60V, 0.2A- *2N7000*	214-1276	9845178	512-2N7000
7	4	R2, R4, R9, R10	Resistor, 1/4W, 1%, 10kΩ		9341110	
8	1	R11	Resistor, 1/4W, 1%, 470Ω		9339531	
9	2	R12, R13	Resistor, 1/4W, 1%, 1kΩ		1457967	
10	2	R16, R17	Resistor, 1W, 10kΩ	214-1276		279-ROX1S10K
11	1	U1	CMOS Quad 2-Input NOR Gate - *CD4001B*	662-9483	1739899	595-CD4001UBE

Table 7.1: List of electronics components necessary for limit detection PCB

- J6.6 $\longleftrightarrow$ J5.7 (*ELMO* Programmable input return).

3. Connect the *ELMO Controller* (*J4* connector) to an auxiliary +24V DC power supply using cable *CBLIC-BAS-J4*. If an auxiliary power supply is not available connect the cable to the +24V DC main power supply.

   - J4.1 $\longleftrightarrow$ +24V DC auxiliary power supply,
   - J4.2 $\longleftrightarrow$ Power supply return.

4. Connect the *ELMO Controller* (*J1 -RJ-45* plug) to a *RJ-45* to *9-pin D-sub* adapter using an ethernet cable. The *9-pin D-sub* connector is normalized according to *CAN*

protocol:

- J1.1 (CAN_H) $\longleftrightarrow$ *9-pin D-sub, pin 7*,
- J1.2 (CAN_L) $\longleftrightarrow$ *9-pin D-sub, pin 2*,
- J1.3 (CAN_{GND}) $\longleftrightarrow$ *9-pin D-sub, pin 3*.

Do not forget to solder a 120Ω termination resistor between pin 7 and pin 2 of *9-pin D-sub* adapter.

1. Connect the *ELMO Controller* (*J8* connector), available on the side of the controller to the motor phases and main power supply.
   - J8.1 (VP+) $\longleftrightarrow$ +24V DC power supply,
   - J8.2 (PR) $\longleftrightarrow$ Power supply return,
   - J8.3 (PE) $\longleftrightarrow$ If available connect to protective earth,
   - J8.4 (PE) $\longleftrightarrow$ motor protective earth (motor body),
   - J8.5 (M1) $\longleftrightarrow$ motor phase 1,
   - J8.6 (M2) $\longleftrightarrow$ motor phase 2,
   - J8.7 (M3) $\longleftrightarrow$ motor phase 3.

There are no constraints in the pinning of M1, M2 and M3.

7.2.3 Connecting the limit detector PCB

J3 on limit detector PCB is the last connector to be affected:

- J3.1 $\longleftrightarrow$ L/D wire (white), lateral sensor,
- J3.2 $\longleftrightarrow$ Q (black), lateral sensor,
- J3.3 $\longleftrightarrow$ - (blue), lateral sensor,
- J3.4 $\longleftrightarrow$ + (brown), lateral sensor,
- J3.5 $\longleftrightarrow$ L/D wire (white), front sensor,
- J3.6 $\longleftrightarrow$ Q (black), front sensor,
- J3.7 $\longleftrightarrow$ - (blue), front sensor,
- J3.8 $\longleftrightarrow$ + (brown), front sensor.

Place jumpers on J4 and J5 connectors between GND and center pin.

A Connecting the *IEPE supply modules M28*

If *MMF IEPE supply modules M28*[5] are connected to audio microphones, the +24V DC main power supply can also by used for this modules.

5 http://www.mmf.de/manual/m28mane.pdf

List of acronyms

AFE Auditory Front-End . . . 8
ALSA Advanced Linux Sound Architecture . . . 20
API Application Programming Interface . . . 27
CAN Control Area Network . . . 31
CPU Central Processing Unit
dof degrees-of-freedom . . . 19
FIFO First In, First Out . . . 21
HATS Head-And-Torso Simulator . . . 3
IEPE Integrated Electronics Piezo Electric . . . 19
PCB Printed Circuit Board
pdf probability density function . . . 59
RPC Remote Procedure Call . . . 9
SLAM Simultaneous Localization and Mapping . . . 19
URDF Unified Robot Description Format . . . 32

Bibliography

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (**1998**), "An Architecture for Autonomy," *Int. Jour. on Robotics Research* **17**, pp. 315–337. (Cited on page 11)

Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Ore-back, A. (**2005**), "Towards component-based robotics," in *IEEE Int. Conf. on Intelligent Robots and Systems*, Tsukuba, Japan. (Cited on page 10)

Bustamante, G., Danès, P., Forgue, T., and Podlubne, A. (**2016**a), "A One-step-ahead Information-based Feedback Control for Binaural Active Localization," in *European Signal Processing Conference (EUSIPCO'2016)*, Budapest, Hungary. (Cited on page 60)

Bustamante, G., Danès, P., Forgue, T., and Podlubne, A. (**2016**b), "Towards Information-Based Feedback Control for Binaural Active Localization," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'2016)*, Shanghai, China. (Cited on page 60)

Bustamante, G., Danès, P., Forgue, T., Podlubne, A., and Manhès, J. (**2016**c), "An Information-based Feedback Control for Audio-Motor Binaural Localization," *Autonomous Robots* Under second review. (Cited on page 60)

Bustamante, G., Portello, A., and Danès, P. (**2015**), "A Three-Stage Framework to Active Source Localization from a Binaural Head," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'2015)*, Brisbane, Australia. (Cited on page 59)

Cadenat, V. (**1999**), "Commande referencee multi-capteurs pour la navigation d'un robot mobile," Master's thesis, Universite Paul Sabatier de Toulouse. (Cited on page 33)

Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (**2010**), *BRIEF: Binary Robust Independent Elementary Features*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 778–792, URL `http://dx.doi.org/10.1007/978-3-642-15561-1_56`. (Cited on page 65)

Corke, P. (**2015**), "Integrating ROS and MATLAB," *Robotics & Automation Magazine, IEEE* **22**, pp. 18–20. (Cited on page 14)

Daniilidis, K. and Eklundh, J.-O. (**2008**), "3-D Vision and Recognition," in *Handbook of Robotics*, edited by B. Siciliano and O. Khatib, Springer-Verlag Berlin Heidelberg, chap. 23. (Cited on page 28)

Grisetti, G., Stachniss, C., and Burgard, W. (**2007**), "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters," *Transactions on Robotics, IEEE* **23**, pp. 34–46. (Cited on page 35)

Isard, M. and Blake, A. (**1998**), "Icondensation: Unifying Low-Level and High-Level Tracking in a Stochastic Framework," in *European Conference on Computer Vision (ECCV'98)*, pp. 893–908. (Cited on page 64)

Labbé, M. (**http://introlab.github.io/find-object 2011**), "Find-Object," `http://introlab.github.io/find-object`. (Cited on page 65)

Lewis, J. (**1995**), "Fast Normalized Cross-Correlation," in *Vision Interface (VI'1995)*. (Cited on page 64)

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (**2010**), "GenoM3: Building Middleware-independent Robotic Components," in *IEEE Int. Conf. on Robotics and Automation (ICRA'2010)*, Anchorage, AK. (Cited on page 11)

Moussy, E., Mekonnen, A., Marion, G., and Lerasle, F. (**2015**), "A Comparative View on Exemplar 'Tracking-by-Detection' Approaches," in *IEEE Intl. Conf. Advanced Video and Signal-based Surveillance (AVSS'2015)*. (Cited on page 64)

O'Kane, J. M. (**2013**), *A Gentle Introduction to ROS*, Independently published, available at `http://www.cse.sc.edu/~jokane/agitr/`. (Cited on page 10)

Rosten, E. and Drummond, T. (**2006**), "Machine Learning for High-speed Corner Detection," in *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*, Springer-Verlag, Berlin, Heidelberg, ECCV'06, pp. 430–443, URL `http://dx.doi.org/10.1007/11744023_34`. (Cited on page 65)

Viola, P. and Jones, M. (**2001**), "Rapid Object Detection using a Boosted Cascade of Simple Features," pp. 511–518. (Cited on page 63)